AR-008-926

DSTO-RR-0008

An Approach To Automated Reasoning About Operational Semantics

A. Cant and M.A. Ozols

19950214 087

An Approach To Automated Reasoning About Operational Semantics

A. Cant and M.A. Ozols

Information Technology Division
Electronics and Surveillance Research Laboratory

ABSTRACT

Research Report

The assurance of the safety or security of critical software rests on a clear understanding of the formal semantics of the programming language used. Operational semantics is the most widely used means of formally defining a language. The need for high levels of assurance, along with the complexity of these definitions for real programming languages, means that tool support is essential for carrying out reasoning about code with respect to the language definition.

In this paper, we describe a generic approach to automated reasoning about the operational semantics of programming languages. As an application of this approach, we describe the construction of an environment for reasoning about programs written in a functional subset of ML. The system we describe (called Elle) captures the formal operational semantics definition of a large subset of Standard ML within the theorem prover Isabelle, and provides some support for the verification of ML programs.

Approved for public release

DSTO-RR-0008

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108

Telephone: (08) 259 6700
Fax: (08) 259 5980
© Commonwealth of Australia 1994
AR-008-926
August 1994

CONTENTS

Page No.

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The work described in this paper is motivated by the general problem of assuring the safety or security of critical software, in which a single error could have disastrous consequences. In such software, the conventional methods of validation (such as testing) are not sufficient to give the required assurance. They must be complemented with *verification* activities which provide formal proofs of the correctness of the code with respect to its (formally specified) critical requirements.

The verification of code requires a clear understanding of the role of the formal semantics (i.e. meaning) of the programming langauge in question. Clearly, knowledge of the precise meaning of a program must be a precursor to any formal statements about the correctness of the program. The formal semantics of the language is 'important even for negotiators and contractors, for a robust program written in an insecure language is like a house built upon sand.' [1]. The work reported on here has arisen directly from the desire to explore the role of formal semantics in the verification process.

The most widely used approach to formal semantics today is that of *operational semantics* [2, 3]. The semantics is given as a number of inference rules which describe under what conditions a language construct will evaluate to a particular value. This method of language definition is a useful guide for the implementer of an interpreter or compiler for the language. It is especially useful for describing the semantics of concurrent languages and process algebras [4, 5], where the denotational semantics may be technically difficult and less easy to mechanise.

The operational semantics definition of a programming language can be quite extensive, and cumbersome to work with. Because of this, tool support is useful for understanding and managing such definitions; it is essential if one wishes to carry out reasoning about programs according to the definition.

In this paper, we describe a generic approach to automated reasoning about the operational semantics of programming languages. A method for capturing operational semantics definitions is described in detail for the Isabelle theorem prover [6].

The best-known example of a language which is *fully* defined via a formal operational semantics is Standard ML [1]. The programming language ML (ML stands for Meta-Language), was originally developed during work on the early theorem prover LCF [7, 8]. Although, to our knowledge, ML has not yet been used in critical software projects, it has nevertheless evolved to the point where it is an important programming language in its own right, expressive enough for many real applications, and with numerous desirable features (such as strong typing, exception handling and modules) which modern software engineering and critical software development require.

In this paper we therefore describe, as a significant application of our approach, the construction of a verification environment, called **Elle**, for reasoning about Standard ML programs. We chose a substantial subset of this language, namely the functional subset of the Core of SML (we shall denote this language by $\mathcal{F}$). The system captures the formal operational semantics definition of $\mathcal{F}$ within Isabelle. Both the static (type-checking or elaboration) and the dynamic (evaluation) semantics of $\mathcal{F}$ are incorporated.

Simple proof procedures can prove $\mathcal{F}$ programs correct by inferring the result of evaluation or elaboration demanded by the definition of SML. The work will benefit those trying to understand the definition, and experts who wish to explore possible modifications and extensions.

The approach described in this paper is applicable to any language defined via its operational semantics. In future work, it is planned to describe the application to the problem of automated reasoning about programs in process algebras such as CCS [9] and CSP [10]. The method is equally applicable to reasoning about properties of formal specifications in languages such as Z, whose semantics can be described by a proof theory such as $\mathcal{W}$ [11].

The rest of this paper is structured as follows. In Section 2, we describe our general approach to reasoning about operational semantics. In Section 3 we give a brief overview of the Isabelle theorem prover [6], and describe in Section 4 how our method is captured within Isabelle. We then give in Section 5 a description of some aspects of set-theoretic reasoning in Isabelle which are necessary for our work. The rest of the paper describes the particular application to reasoning about Standard ML (SML). In Section 6 we describe the structure of the Definition of SML. The following sections (Sections 7 to 11) give a detailed description of the **Elle** system. Finally, Section 12 discusses the results and give suggestions for further work.

2 Reasoning About Operational Semantics

2.1 Introduction

An operational semantics definition for a programming language is an instance of a *transition system* [3], which is a pair $(\Gamma, \Rightarrow)$, where Γ is a set of configurations, and $\Rightarrow$ is a relation on Γ, with the interpretation that $\gamma_1 \Rightarrow \gamma_2$ means that γ_1 evaluates to γ_2.

Suppose that L is a context-free language [12], described by a context-free grammar $G = (V, T, P)$, where:

- V is a finite set of *nonterminals*
- T is a finite set of *terminals* $(V \cap T = \emptyset)$
- P is a finite set of *productions*, each production π being of the form $v \longrightarrow \alpha$, where $v \in V$ and α is a string of symbols from $(V \cup T)^*$

We shall further suppose that we have a set $I = \{id_1, ..., id_k\}$, with $I \subseteqq T$, of *identifier* symbols.

The (operational) semantics of the language L is given as follows. Firstly, the relevant *semantic objects* must be prescribed. These will be mathematically well-understood sets or functions, and are the domains in which the meaning of the language will be defined. We have a number of *simple semantic objects*, from which certain *compound semantic* *objects* are built, via standard set- or function-theoretic operations:

$A \times B$ (Cartesian Product)

$A \overset{fin}{\to} B$ (Finite Maps)

$A \sqcup B$ (Disjoint Union)

$f \dagger g$ (Overwriting)

For each production π of G, with π of the form $v \longrightarrow \alpha$, we associate a formula $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$ called a *sequent*. The interpretation of this sequent is as follows. When we *evaluate*, i.e. compute the meaning of, the phrase α against the semantic background represented by the collection $\mathcal{C}$ of semantic objects, we obtain the result $\mathcal{C}'$. In general $\mathcal{C}'$ itself can be a collection of semantic objects.

The circumstances under which we can infer the truth of this sequent are described by a collection of inference rules R_i $(1 \leq i \leq n)$ of the form:

$$R_i \equiv \frac{Cond_i^1 \quad \ldots \quad Cond_i^{k_i}}{\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'}$$

where the premises $Cond_i^j$ are either sequents or *side-conditions*, i.e. other formulae involving semantic objects.

2.2 Requirements

In constructing a system for machine-assisted reasoning about programs in the language *L*, our aim is to capture the syntax and formal operational semantics in as natural a way as possible in a suitably powerful theorem prover.

There are three general kinds of proof activity which we wish to be able to automate, as follows.

- *Evaluation of Phrases*. Here we wish to discover the result of an evaluation in accordance with the semantics. We do this by asserting the goal to be proved in the form

  $$\mathcal{C} \vdash_v \alpha \Rightarrow ?\mathcal{C}'$$

  where $?\mathcal{C}'$ is a scheme variable for the as-yet unknown result of evaluation. It will be instantiated during the proof.

- *Proofs of Correctness*. Here the goal is of the form

  $$\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$$

  with the result already believed known; other sequents or side conditions may also be involved as assumptions in the proof.

- *Reasoning About Equivalences*. We wish to be able to establish equivalences, where we say that two phrases α and α' are *equivalent* if they evaluate to the same result in every semantic background, i.e.

  $$(\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}') \Leftrightarrow (\mathcal{C} \vdash_v \alpha' \Rightarrow \mathcal{C}')$$

We shall now address specific requirements which need to be met by the theorem prover.

2.2.1 Language Requirements

2.2.1.1 Genericity

The method must be generic, i.e. the basic idea should be applicable across a range of languages.

2.2.1.2 Concrete Syntax

It is highly desirable that the theorem prover chosen should allow, as far as possible, the concrete syntax of L to be faithfully represented. The same requirement holds for semantic objects. Reasoning about language semantics is already a difficult and challenging activity; it should not be impeded by having proofs presented in an unfamiliar or even unreadable syntax. Having a reasonably clean concrete syntax, which is also strictly maintained during a proof, is a great aid to understanding.

2.2.1.3 Derived Forms

A language will often have a number of *derived forms*, namely syntactically "new" language constructs which are mere surface syntax for more primtitive constructs. (Such derived forms are important in SML). There are no inference rules for these derived forms; they will always be reduce to the primitive constructs. The prover should support the use of derived forms.

2.2.2 Semantic Requirements

2.2.2.1 Support for Semantic Objects

An expressive (and preferably familiar) object-logic will be needed to capture the basic properties of semantic objects.

2.2.2.2 Support for Meta-Level Reasoning

The above inference rules are used in practice both as *introduction rules*, in which we attempt to establish the truth of the sequent $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$, or as *elimination rules*, in which this sequent is given as an assumption, and we reason that it must have been inferred via R_i (for some i). The latter is an instance of *meta-level reasoning*. If we wish to be able to carry out the most general reasoning about programs in the language L — such as proofs of correctness, transformations, equivalences etc — then our method will need to be able to express inference rules in such a way that the effect of applying the corresponding introduction and elimination rules can easily be achieved.

2.2.2.3 Logical vs Program Variables

We shall need to be able to mix program variables (which are part of the language) and logical variables (used for quantifying over language phrases and semantic objects), and have a way of distinguishing between them.

2.2.3 Proof Requirements

2.2.3.1 Tactic Language

The theorem prover should support goal-directed proofs by means of *tactics*, and it should be easy to built new tactics from existing ones. Subtle proof procedures are necessary; thus, the language of tactics should also be sufficiently expressive.

2.2.3.2 Powerful Proof Heuristics

Ideally, the theorem prover should have built-in powerful automated proof methods which apply Artificial Intelligence techniques of search, attempts at proofs by induction etc. These are called *heuristics*.

2.2.3.3 Backtracking Search

The language inference rules often allow a particular sequent to be inferred in more than one way. In a particular goal-directed proof, the chosen theorem prover will need to be able to manage multiple (possibly infinitely many) proof states, and to back up if the wrong branch of the proof-tree is chosen.

2.2.3.4 Answer Extraction

It will be important to have the facility (as Prolog does) for *answer extraction* during a proof: for example, we may wish to establish the truth of the sequent $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$ without knowing in advance all of the semantic objects $\mathcal{C}$ and $\mathcal{C}'$.

2.2.3.5 Recursive Domains

Ideally, the prover should support reasoning about recursive functions and datatypes, allowing the automatic production of induction theorems, cases theorems etc.

2.2.4 User Interface

Clearly it is desirable to use a theorem prover with a user-interface that makes the theory building and theorem-proving tasks as straightforward as possible. Most theorem proving tools have primitive user-interfaces; however, recent advances in graphic user interface technology have meant that simple window-based interfaces are under development for most provers.

2.3 Discussion

Based on the above requirements, we surveyed a number of theorem provers, of which four were chosen as indicative: namely, Isabelle [6], Mural [13], HOL [14] and EVES [15]. (Prolog has been used for the purpose [16]. However, we have ruled it out, because it fails to meet most of the above requirements.)

The following table summarises whether or not the requirements are met for these provers. It can be seen from the table that, although HOL is a highly expressive system

Table 1 Properties of Theorem Provers

	Isabelle	Mural	HOL	EVES
Generic	Yes	Yes	No	No
Concrete Syntax	Yes	Yes	No[1]	No
Derived Forms	Yes	No	No	No
Suitable Object-Logic	Yes (ZF or HOL)	Yes (VDM Logic and Data)	Yes (HOL itself)	Yes (Verdi)
Meta-Level Reasoning	Yes	Yes	Yes	Yes
Logical Variables	Yes	Yes	Yes	Yes
Tactic Language	Yes, flexible	Yes, not so usable	Yes, flexible	No
Proof Heuristics	Fair	Fair	Limited	Very Good
Search	Yes	Yes (rarely used)	No	Yes (built-in)
Answer Extraction	Yes	No	No	No
Recursive Domains	Yes	Yes	Yes	Yes
User Interface	Basic	Windows	Basic	Basic

with a large user base, it does not meet our requirements for features such as concrete syntax, derived forms, answer extraction and backtracking search. Likewise, the EVES system, in many ways the 'state-of-the-art' in powerful theorem-proving tools, again does not have the facilities for concrete syntax, derived forms and answer extraction, and its search facility is not customisable. Thus, our choice of theorem prover can really be narrowed down to two possibilities, namely Isabelle and Mural. Mural has an excellent user interface which allows concrete syntax. This alone would make it a serious candidate; however, it lacks the facilities of derived forms, answer extraction, expressive tactic language, and the search facility is rudimentary.

Isabelle was chosen for our work, because it meets all the essential requirements. In the next Section, we shall give an overview of Isabelle.

[1] Note that HOL 90, an implementation in SML, does allow concrete syntax

3 Isabelle

3.1 Introduction

Isabelle has been under development by Larry Paulson and collaborators at the University of Cambridge since 1986 [6, 17, 18]. It is a member of the LCF family [8] of tactical theorem provers, and is written in Standard ML.

Isabelle is a generic prover: the logic of discourse (*object logic*) may be defined by the user, or chosen from one of the object logics provided with the system. Isabelle also has an expressive *meta-logic*, in which the inference rules and axioms of these object logics can be formulated. Isabelle allows concrete syntax, and supports derived forms via user-provided parse and print translations.

Semantic objects can be comfortably described within various possible object logics: the most natural choice is Isabelle's Zermelo-Fraenkel (ZF) Set Theory. Meta-level reasoning (i.e. the two-way nature of operational semantics inference rules) can easily be captured within these logics, as will be seen later. Logical and program variables can be handled.

Isabelle supports both forwards proof, and backwards (goal-directed) proof using tactics; it has a subgoal package, for manipulating proofs interactively. Isabelle lacks the powerful proofs commands found in the EVES system, but it is straightforward to write appropriate proof procedures using the Isabelle tactic language. New tactics can be constructed from existing ones by means of *tacticals*. Various search tactics can be constructed. Answer extraction during proofs is made possible by Isabelle's *scheme variables*.

Recursive datatypes and functions can be defined by means of the machinery of least fixed points, developed recently by Paulson for the ZF and HOL logics [19].

A window-based interface for Isabelle is not yet generally available, but an interface based on the Centaur system [20] has been developed by Laurent Théry at the University of Cambridge [21]. A window-based interface called XIsabelle, implemented using the PolyML/Motif [22] system, is also being developed at DSTO by the authors. This interface offers theory browsing facilities, and provides advice to the user about applicable matching rules during interactive proofs. The interface is described in [23].

In this section we shall briefly describe the various aspects of Isabelle (meta-logic, object logics and their theory files, forwards and backwards proof, tactics and tacticals).

3.2 The Meta-Logic

The meta-logic must be compact, but expressive enough to be able to formulate the rules and axioms for object logics. Isabelle's meta-logic is intuitionistic higher-order logic with universal quantification and equality, and a type system with order-sorted polymorphism. Pure Isabelle implements the meta-logic.

The tables below show the constructs used in the meta-logic (*types, terms and formulae*). Examples appear in the rest of this Section.

Table 2 Notation for Types

Notation	Description
$\tau :: C$	class constraint
$\sigma \Rightarrow \tau$	function type
$[\sigma_1, \ldots, \sigma_n] \Rightarrow \tau$	curried function type
$(\sigma_1, \ldots, \sigma_n)\ tyop$	type construction

Table 3 Notation for Terms (typed λ-calculus)

Notation	Description
$t :: \sigma$	type constraint
$\lambda x.\phi$	meta-abstraction
$t(u_1, \ldots, u_n)$	application

Table 4 Notation for Meta-Formulae

Notation	Description
$a \equiv b$	meta-equality
$\phi \Rightarrow \psi$	meta-implication
$[\phi_1; \ldots \phi_n] \Rightarrow \psi$	nested implication
$\bigwedge x.\phi$	meta-quantification

3.3 Object Logics and Theories

An object logic is an ML object of type **theory**. The axioms and rules are of type **thm**. Isabelle comes provided with a number of object logics, including First Order Logic (FOL), Zermelo-Fraenkel Set Theory (ZF) and a version of Higher-Order Logic (HOL).

Isabelle works with inference rules expressed in a natural deduction style. Each logical connective has, in general, elimination and introduction rules of inference. For example, Table 5 shows examples of these rules for conjunction and implication in first-order logic.

Table 5 Examples of Inference Rules

	Introduction (I)	Elimination (E)	
Conjunction	$\dfrac{A \quad B}{A \wedge B}$	$\dfrac{A \wedge B}{A}$	$\dfrac{A \wedge B}{B}$
Implication	$\dfrac{[A] \quad B}{A \supset B}$	$\dfrac{A \supset B \quad A}{B}$	

Note that the rule for implication elimination is just *modus ponens*. In the meta-logic, it is expressed (using nested meta-implication) as follows:

$$[A; A \supset B] \Rightarrow B$$

In Isabelle, theories are most easily specified by means of *theory files* [6]. A theory file declares the parents of the theory in question, new classes and types and new constants along with their types and concrete syntax. It then gives definitions (via meta-equality), axioms and inference rules for the logic. The theory file may also include some derived forms, given by means of parse and print translations. Once these translations have been provided, we can use the derived forms freely in goals, and they will be correctly dealt with.

3.4 Forwards Proof

Isabelle allows new theorems[2] to be created by *resolution*. In general, suppose we have two theorems of the form:

$$[A_1; ...; A_m] \Rightarrow A$$
$$[B_1; ...; B_n] \Rightarrow B$$

where A unifies with B_i, so there is a substitution s such that $s(A) = s(B_i)$. Then, by resolution, we have a new theorem:

$$s([B_1; ...; B_{i-1}; A_1; ...; A_m; B_{i+1}; ...; B_n] \Rightarrow B)$$

For example, from the two theorems

$$P \wedge Q \Rightarrow P \text{ (conjunct1)}$$
$$P \Rightarrow P \vee Q \text{ (disjI1)}$$

we can use resolution to obtain the theorem :

$$P \wedge Q \Rightarrow P \vee Q$$

2 Note that in Isabelle 'theorem' includes inference rules and definitions in the object-logic.

3.5 The Subgoal Package

Isabelle carries out goal-directed proofs, and contains a subgoal package to assist with interactive proof. A *proof state* consists of a *goal*, along with a number of subgoals whose validity establishes that of the goal. The subgoals can be thought of as proof obligations. Diagrammatically we display a proof state as follows:

$$\frac{\mathit{initial\ goal}}{\mathit{subgoal}_1 \ \ldots\ \mathit{subgoal}_n}$$

When we set a goal in Isabelle we have as our initial proof state

$$\frac{\mathit{goal}}{\mathit{goal}}$$

in which there is a single subgoal identical with the original goal. A proof state with no subgoals is a proof of the original goal.

3.6 Basic Tactics

Proof states are transformed to new states by means of the application of *tactics*. In Isabelle a tactic may fail, or return one or more new proof states, possibly a lazy infinite list.

In general, if T is a tactic, and ϕ a proof state, then the result $T\phi$ of applying T to ϕ is written as a list to capture the various alternatives:

$$\begin{aligned}
&T\phi = [\ ] \quad \text{(failure)}\\
&T\phi = [\psi] \quad \text{(unique result)}\\
&T\phi = [\psi_1, \psi_2, \psi_3, \ldots] \quad \text{(multiple outcomes)}
\end{aligned}$$

If the tactic succeeds, the head of this list is the active proof state, and is usually presented with all of its subgoals shown. Many tactics act on a number of subgoals, automatically instantiating variables and renumbering the subgoals as appropriate.

Pure Isabelle has a number of commonly used basic tactics (object logics also have their own special purpose tactics). We shall discuss the most important of these.

Recall that Isabelle emphasises the natural style of reasoning; correspondingly, most proof steps are carried out backwards reasoning using inference rules of the logic. This is called *resolution*. Isabelle provides a single ML function to do this.

The basic resolution tactic is `resolve_tac thms i`. This tactic tries each theorem in the list `thms` against subgoal `i` of the proof state, until a rule is found whose conclusion can unify with the subgoal. For a given rule, say

$$[B_1; \ldots; B_k] \Rightarrow B$$

and subgoal

$$[A_1; \ldots; A_n] \Rightarrow A$$

where A can unify with B under the substitution s, resolution replaces A by the instantiated premises $\overline{B_1}, ..., \overline{B_k}$, producing a new state with the following subgoals:

$$s([A_1; ...; A_n] \Rightarrow \overline{B_1})$$

$$\ldots$$

$$s([A_1; ...; A_n] \Rightarrow \overline{B_k})$$

in which the instantiations resulting from the substitution s have been made.

Subgoals frequently change their appearance as instantiations propagate throughout the proof tree.

Multiple outcomes can arise in two ways. Firstly, unification in Isabelle is higher-order unification (i.e. solving equations in the typed λ−calculus with respect to α, β and η conversion [24]). There is no most general unifier, and so there can be more than one higher-order unifier. Secondly, more than one theorem may be able to be resolved with the goal. The tactic will fail if none of the rules can be unified.

Another fundamental tactic is `assume_tac i`, which tries to solve subgoal `i` by assumption (again, this may involve unification).

Reasoning about definitions and deriving new rules is facilitated by a number of rewriting tactics. For example, `rewrite_goals_tac thms` uses the given definitional theorems for rewriting subgoals. Excessive rewriting is not good Isabelle style (and can be expensive); the preferred strategy is immediately to derive elimination and introduction inference rules for any new construct, and thereafter to use these new rules in resolution steps.

Isabelle also allows conditional object-level rewriting: for example, `SIMP_TAC ss i`: uses a given set ss of object-level simplification rules, and rewrites subgoal i. Rewriting can be useful for one-off proofs, but is slow compared to resolution.

Isabelle also has answer extraction available, via so-called *scheme variables*. These variables can be part of a goal; as tactics are applied the scheme variables may be instantiated during the proof.

3.7 Tacticals

The power of a tactical theorem prover rests in the ability to combine tactics to build new ones, by means of *tacticals*. A selection of basic tacticals is as follows:

`tac1 THEN tac2` (sequencing)
`tac1 ORELSE tac2` (choice)
`REPEAT tac` (iteration)
`DEPTHFIRST pred tac` (search)

The tactic `tac1 THEN tac2`, applied to the proof state ϕ, first computes `tac1`(ϕ), giving some list $[\psi_1, \psi_2, ...]$ of proof states, and then applies `tac2` to each of these states, giving as output the concatenation of the sequences `tac2`(ψ_1), `tac2`(ψ_2).

The tactic `tac1 ORELSE tac2` is a form of choice: it first computes `tac1`(ϕ). If this is non-empty, it is returned as the result; otherwise, `tac2`(ϕ) is returned.

The tactic `REPEAT tac` first computes `tac`(ϕ). If this is non-empty, then the tactics recursively applies itself to each element, concatenating the results. Otherwise, it returns the singleton list $[\phi]$.

The tactic `DEPTH_FIRST pred tac` performs a depth-first search for a proof-state satisfying `pred`. Usually `pred` is taken to be "no subgoals", so that the tactic will search for a proof of the original goal.

4 Operational Semantics in Isabelle

Isabelle is ideal for reasoning about operational semantics, because it has been designed for natural deduction, and works well with derived inference rules. If we regard our operational semantics as the rules for a logic, then we can quickly construct powerful proof procedures in Isabelle which allow reasoning about quite complicated programs.

There will, in general, be a number of ways of capturing the operational semantics of a given programming language within Isabelle. These are not just matters of implementation detail, but should be regarded as 'design' issues to be resolved for the logic. They are important, because they influence the way that the user reasons about programs in the language. In what follows, we make some general observations about these issues.

4.1 Language Syntax

The first issue is how we represent the syntax of the context-free language L (introduced in Section 2.1) within a new Isabelle theory. First of all, for each non-terminal $v \in V$, we declare a new type $Ty(v)$. It is also useful (as we shall see later) to assign each identifier symbol in I to the same type $Ty(id)$. The type-checking discipline imposed by Isabelle will carry out the function of syntax-checking for sentences in the language; this prevents a lot of errors.

Now suppose that π is a production, of the form $v \rightarrow \alpha$, with

$$\alpha = \Gamma_1 w_1 \ldots \Gamma_n w_n$$

where each Γ_i is either a non-terminal or an identifier symbol , and each w_i is a string of other (non-identifier) terminal symbols. Then this production is represented in the theory by the new constant $C(\pi)$, with curried function type $[Ty(\Gamma_1), \ldots, Ty(\Gamma_n)] \Rightarrow Ty(v)$, and whose concrete syntax is given by the pattern of strings of non-identifier terminal symbols occurring in α.

This scheme may seem elaborate (especially the treatment of identifiers), but in practice it is very simple to set up. An example should help to explain the technique. Suppose that *plus* is the production

$$Exp \rightarrow Exp + Exp$$

Then the theory will contain a constant which can be denoted by $C(plus)$, which has type $[Ty(Exp), Ty(Exp)] \Rightarrow Ty(Exp)$. The concrete syntax is given by $\_ + \_$, where the underscores are placeholders for the arguments of $C(plus)$.

4.2 Semantic Domains

The second issue is how we represent semantic objects within Isabelle. As discussed earlier (in 2.1), these are familiar set and function theoretic objects, well-known in specification languages such as Z [25] and VDM [26]. The decision we need to make is how to reason about such constructs within the context of proving properties of programs in the language L. Two broad approaches are possible, which we shall now describe.

The first approach is to do what is required from scratch, i.e. forget about existing Isabelle logics, and represent all semantic objects, such as environments and record values, as new *types*; provide a new concrete syntax for them, and postulate the necessary introduction and elimination rules to allow the proof of goals involving these constructs. The advantages of this approach are that the rules are natural and easy to understand, and the proof procedures can be reasonably simple resolution tactics. However, the disadvantages are that a considerable number of rules are required to capture the meaning of quite familiar objects. Relying heavily on inference rules in this way increases the chance of errors, and could compromise the soundness of the system. Moreover, some constructs (such as type closures in Standard ML) are difficult to handle in a straightforward manner.

The second approach is to make use of an existing Isabelle base logic which already contains as many of the required set and function theoretical constructs as possible. Any semantic objects which are not part of the base logic can be defined (using meta-level rewriting) in terms of simpler objects, provided that the logic is expressive enough. The advantages of this approach are that we do not have to 'reinvent the wheel', and design the relevant rules for reasoning about these constructs. We can also make use of existing tactics, in particular the appropriate simplifier (rewriting engine), if available. This approach is more likely to yield a sound system than the above because the built-in Isabelle object logics are constructed using well-known axiomatisations. The main disadvantage is that the axiomatic formulation of the logic may be complicated, and take some time and effort to learn before proofs can be attempted. Also, the concrete syntax for the base logic may clash with that of the language being studied.

Our philosophy is to make the correspondence with the formal definition of the semantics of L as close as possible, and to rely on Isabelle's extensive support for reasoning about familiar semantic domains. Therefore, we believe that the second approach is preferable.

Having decided on this approach, we can distinguish three different kinds of rule in our theory of semantic objects. These will now be described.

4.2.1 Meta-Level Rewrite Rules

Meta-level rewrite rules use meta-equality to define a new construct A in terms of other constructs. Such rules take the form

$$A(t_1, ..., t_n) \equiv f(t_1, ..., t_n)$$

When $A(t_1, ..., t_n)$ occurs in a formula, it can be expanded according to its definition by applying meta-level rewriting (sometimes called *unfolding* — the reverse operation is called *folding*). It is possible for the right hand side of the rule to involve A, so that recursive definitions can be made. This must be done with care, in order to ensure that no inconsistencies are introduced into the logic.

An example of this is the definition of variable environments for elaboration (see Section 10):

$$VarEnv == (SUM\ x : \{\,Var, Con, ExCon\}.x) \rightarrow TypeScheme$$

which expresses the fact that a variable environment is a finite map from the disjoint union of the classes of variables, value constructors and exception constructors to the set of typeschemes.

4.2.2 Introduction/Elimination Rules

These have been mentioned earlier (see Section 2). Rules of the form:

$$[A_1; ...; A_n] \Rightarrow A$$

can be regarded as introduction rules for A, or as an elimination rule for one of the assumptions A_i. They may be derived from definitions (see above), or even postulated directly. An example of such a rule is

$$[f\ : A \rightarrow B;\ g\ : A \rightarrow B] \Rightarrow f \dagger g\ : A \rightarrow B$$

which says that if f and g are functions, then f overwritten by g is also a function (see 5.1)

4.2.3 Object-Level Rewrite Rules

Object-level rewriting can be used to transform terms with respect to some relation $\gg$ which is both reflexive and transitive. Most commonly, the appropriate relation is either logical equivalence or equality in First Order Logic and its extensions.

Rules of the form

$$[A_1, ..., A_n] \Rightarrow P \gg P'$$

can be used for (conditional) rewriting, making use of the simplifier. The intention is that any subterms P of a given term can be rewritten at the object-level to P', provided the pre-conditions $A_1, ..., A_n$ are met. For example, in our theory we have the rules

$$f \dagger 0 = f \text{ (for functions)}$$
$$A\ Un\ 0 = A \text{ (for sets)}$$

These can be used as object-level rewrite rules, and taken account of by the simplifier, if required. Often, a special tactic will be needed for solving the subgoals arising from conditional rewrites.

Note that the new Isabelle simplifier is less general than described here, but considerably faster. It is still sufficiently general to cover all our uses in the **Elle** system.

4.3 Language Semantics

Finally, we need to represent the semantics of the language L itself within Isabelle. Language sequents of the form $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$ will be represented by formulae.

To represent the language inference rules in Isabelle, we recall the need to capture meta-level reasoning. We express the inference rules in ZF in the following form:

$$\begin{aligned}&\Bigl(\bigl(\exists x \in Conds_1 \backslash \mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'.Cond_1^1 \ \wedge \ \ldots \ \wedge Cond_1^{k_1}\bigr)\vee \\ &\ldots. \\ &\bigl(\exists x \in Conds_n \backslash \mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'.Cond_n^1 \ \wedge \ \ldots \ \wedge Cond_n^{k_n}\bigr)\Bigr) \\ &\Leftrightarrow \mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'\end{aligned}$$

where $\exists x \in Conds_i \backslash \mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$ denotes existential quantification over all the semantic variables present in $Cond_i^1, \ldots, Cond_i^{k_i}$ but not in the conclusion $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$ of the rule.

Of course, from the above inference rules we can derive both introduction and elimination rules for $\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'$. We can automate this process, by means of the ML functions:

```
mk_intr_rule : thm -> thm list
get_intr_rules : theory -> string * string list list -> thm list
mk_elim_rule : thm -> thm
```

The function `mk_intr_rule` produces the introduction rules:

$$\frac{Cond_1^1 \ \ldots \ Cond_1^{k_1}}{\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'}$$

$$\ldots.$$

$$\frac{Cond_n^1 \ \ldots \ Cond_n^{k_1}}{\mathcal{C} \vdash_v \alpha \Rightarrow \mathcal{C}'}$$

The function `get_intr_rules` is somewhat similar, but takes an extra argument which, for each of these resulting rules, allows one or more equality substitutions to be made where the premises permit. The function `mk_elim_rule : thm -> thm` is used to derive elimination rules; these will not be discussed in this paper.

5 Set-Theoretic Reasoning in Isabelle

We have chosen to build our new logics on top of Isabelle's Zermelo-Fraenkel Set Theory (ZF), which is an extension of First-Order Logic (FOL). This theory, being concerned with establishing the familiar properties of sets and functions from primitive axioms, is a natural choice. An alternative would have been to use Isabelle's Higher-Order Logic (HOL). We chose ZF because it is simpler than HOL. Its type system is weaker than that of HOL (everything in ZF is a set — and can have only one type), but this can be an advantage when we wish to overload notation for semantic objects without getting distracted from the constructs of real concern — syntax and semantics of the language.

Isabelle's ZF [6] is the work of Martin Coen, Philippe Noel and Lawrence Paulson, and is now a highly developed theory, with an enormous number of derived rules, and a sophisticated simplifier. Further work by Paulson has demonstrated the usefulness of ZF as a computational logic, i.e. it provides simple but sufficient machinery for reasoning about recursive functions and datatypes [19]. The reader should consult these references for more details of the Isabelle implementation of ZF.

Most of the notation we shall use is fairly standard. We note that elements of the disjoint union

$$A_1 \sqcup \ldots \sqcup A_n$$

are pairs of the form

$$\langle A_i, x_i \rangle \;\; where\; x_i \in A_i \;\; (1 \leq i \leq n)$$

Thus we can identify which of the sets A_i the elements belongs to by using the name of A_i itself as a label. (This works provided the A_i are all mutually distinct).

For our work, it was necessary to derive a number of extensions of Isabelle's ZF in order to allow reasoning about semantic objects. We shall now describe these extensions.

5.1 Funtion Overwriting

An important operation in the semantics of a language such as SML is that of *overwriting* one function with another. Many of the language inference rules involve overwritten variable environments. There are a number of ways of defining this operation; we adopt the following, as being the most useful for subsequent development. If f and g are functions, then we define

$$f \;\dagger\; g \equiv \{p \in f \mid \neg fst(p) \in domain(g)\} \;\cup g$$

(This definition also makes sense if f is a relation). Thus the bindings in f are overwritten by those in g, for example

$$\{\langle x, 1\rangle, \langle y, 2\rangle, \langle z, 3\rangle\} \dagger \{\langle x, 2\rangle, \langle w, 5\rangle\} = \\ \{\langle x, 2\rangle, \langle y, 2\rangle, \langle z, 3\rangle, \langle w, 5\rangle\}$$

Various kinds of semantic goals involving functions are possible. The most common are:

$$f \dagger g = ?h$$
$$(f \dagger g)\,(x) = ?a$$
$$f\;(x) = ?a$$

in which each right-hand side is yet to be instantiated, and where f and g are concrete functions. We need an efficient technique for dealing with these goals in such a way that, at the end of a proof, all terms involving function overwrites and applications will be fully evaluated.

To achieve this, we derive a number of rules about overwrite. First of all, we have the introduction rules:

$$\frac{\langle x, a\rangle \in g}{\langle x, a\rangle \in f \dagger g} \quad (\dagger \mathrm{I1})$$

$$\frac{\langle x, a\rangle \in f \quad \neg x \in domain(g)}{\langle x, a\rangle \in f \dagger g} \quad (\dagger \mathrm{I2})$$

Overwrites can be computed directly via

$$\frac{\langle x, b\rangle \in g}{cons(\langle x, a\rangle, f) \dagger g = f \dagger g} \quad (\mathrm{cons} \dagger \mathrm{I1})$$

$$\frac{\neg x \in domain(g)}{cons(\langle x, a\rangle, f) \dagger g = cons(\langle x, a\rangle, f \dagger g)} \quad (\mathrm{cons} \dagger \mathrm{I2})$$

A number of other useful rules have been derived, including:

$$\frac{f \in A \to B \quad g \in C \to D}{f \dagger g \in domain(f \dagger g) \to range(f \dagger g)} \quad (\dagger - \mathrm{type})$$

$$f \dagger (g \dagger h) = (f \dagger g) \dagger h \quad (\dagger - \mathrm{assoc})$$

$$\emptyset \dagger f = f \quad (\dagger\emptyset - \mathrm{left})$$

$$f \dagger \emptyset = f \quad (\dagger\emptyset - \mathrm{right})$$

$$domain(f \dagger g) = domain(f) \cup domain(g) \quad (\mathrm{domain}\dagger)$$

These rules are immediately suitable for use in object-level rewriting, and semantic goals can indeed be attacked in this way. However, the Isabelle simplifiers are not as efficient as one would like, and, after much experimentation, we discovered that it was consistently more efficient in the **Elle** system to use versions of these rules suitable for resolution. These rules have the suffix "-equality", and are trivial to derive. An example is

$$\frac{\langle x, b\rangle \in g \quad f \dagger g = h}{cons(\langle x, a\rangle, f) \dagger g = h} \quad (\mathrm{cons} \dagger \mathrm{I1} - \mathrm{equality})$$

The conclusion of this rule will now match any subgoal of the form

$$f' \dagger g = ?h$$

in which f' is of the form $cons(\langle x, a\rangle, f)$ or is a scheme variable.

5.2 Handling Distinct Variables

To compute, say,

$$\{\langle x, a\rangle\} \dagger \{\langle y, b\rangle\} = \{\langle x, a\rangle, \langle y, b\rangle\}$$

it is tacitly assumed that x and y are distinct variables. However, Isabelle has no way of knowing this, and, use of the overwrite rules will yield a subgoal asserting that $\neg x = y$. To avoid the problem of making tiresome and lengthy assertions of inequalities, we use a scheme designed to provide a set of "reserved" variable names which are guaranteed to be mutually distinct. This is done by 'tagging' names with a unique binary number. This is implemented in the theory `Tags`, which is described in the Appendix.

6 The Definition of Standard ML

Readable introductions to Standard ML (SML) can be found in the books by Paulson [27] and Wikstrom [28]. Here we shall just highlight the main features of the language:

- It is a functional language — functions are first-class objects and can be passed as arguments to other functions, or returned as values.
- It is statically-scoped: identifiers are associated with values according to where they appear in the program text (and not on the run-time behaviour of the program).
- It is a strongly typed language: every ML expression has a statically-determined type.
- It is polymorphic — type expressions may contain type variables, allowing for functions to be defined on a class of arguments of different types.
- It has facilities for abstraction — the user can define new abstract data types and hide the details of their implementation from functions which make use of them.
- It has a modules facility, allowing the grouping of large ML programs into separate units which can be separately compiled.
- It has an exception trap mechanism, to allow the uniform handling of user and system-generated exceptions.
- It has a formal semantics — the language definition of SML [1] is expressed in terms of operational semantics

The formal semantics of Standard ML has been given by Milner, Harper and Tofte [1], (referred to in what follows as 'the Definition'). The companion Commentary [29] gives some insight into the Definition.

The Definition presents the syntax for SML (both for the Core language and the Modules System), introducing various identifier and phrase classes.

The static semantics (i.e. type-checking, or elaboration of phrases) involves a rich set of simple and compound semantic objects (such as environments, types etc). Elaboration of a phrase is expressed by a sequent of the form

$$C \vdash \mathit{phrase} \Rightarrow \mathit{result}$$

where typically C is a context, and the result may be a type or a type environment. The 102 inference rules capture all the possible inferences among these sequents.

The dynamic semantics (evaluation of phrases) is given a similar, but separate, treatment. The fact that evaluation and elaboration can be dealt with independently is an important aspect of SML. Often the same terminology gets used in both the static and dynamic semantics but has a different meaning in each case, such as "variable environment". Static and dynamic semantics meet at the level of programs, where the evaluation of a program is only carried out if it elaborates successfully.

Standard ML has a number of phrase classes which are derived forms. For example, the program phrase `case` *exp* `of` *match* is defined to be the more primitive language expression (`fn` *match*)(*exp*). Other examples of derived forms are `if`, `andalso` and `orelse` as well as lists and tuples. Inference rules only need to be given for the phrases in the so-called 'bare' language.

Because of their formal nature, and the size of the language, the Definition and Commentary are not light reading, and are aimed at implementers and ML experts more than the general reader. The Definition allows one in principle to explore language semantics, but detailed proofs done on paper using all the 196 inference rules are far too laborious: we believe that machine support is essential to be able to do this.

In the following Sections, we shall describe our approach to reasoning about SML within the Isabelle prover.

7 Overview of The Elle System

At present, we have built a system which captures the syntax and semantics of a substantial subset of Standard ML, namely the pure functional (side-effect free) subset of the Core Language. This subset will be denoted by $\mathcal{F}$. The subset includes pattern matching, functions as first-class objects and recursion. We have excluded imperative features such as reference variables, assignments, and exceptions, as well as the modules system. However, what remains is still an extremely rich language.

In the spirit of the Definition [1], elaboration (static semantics) and evaluation (dynamic semantics) are treated separately; this is reflected in the design of the system. Thus, separate Isabelle theories are maintained: one for elaboration and one for evaluation. They have in common the syntax of $\mathcal{F}$ itself.

The syntax, semantic objects and inference rules of $\mathcal{F}$ are captured in a series of new (typed) Isabelle object logics, or theories, built on top of ZF. The various semantic objects, such as values and environments are themselves given an appropriate concrete syntax, and their properties described by means of inference rules within the logic $\mathcal{L}$ for the various operations defined on these domains.

In this section, we shall describe the structure of the **Elle** system. The theory hierarchy of the system is shown in Figure 1.

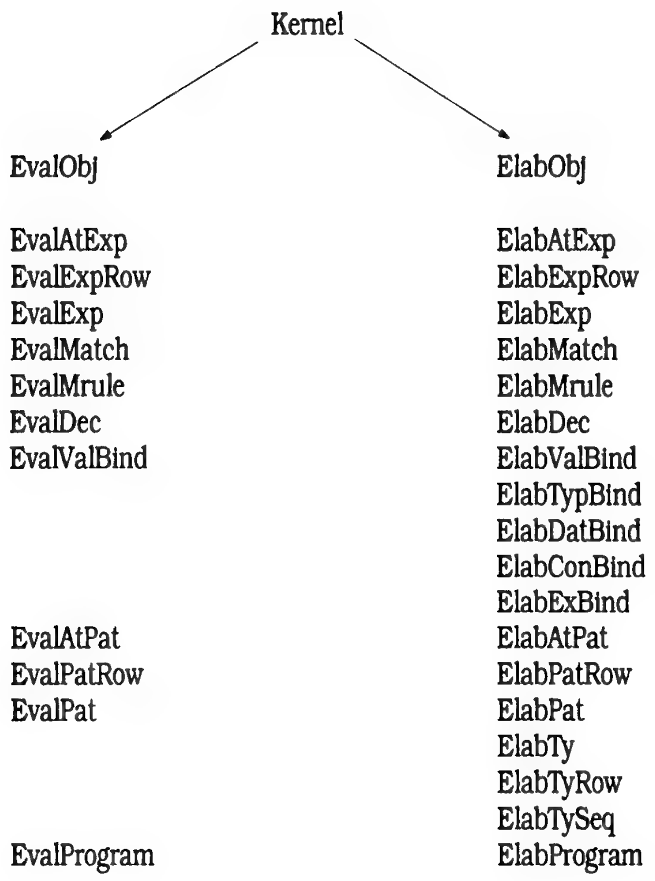

Figure 1 Structure of the **Elle** System

8 The Kernel

The *Kernel* of the system is an ML image which consists of all the Isabelle machinery (theories, derived rules etc) which is common to both elaboration and evaluation. It includes the extensions `Overwrite` and `Tags` of ZF set theory which were described earlier in Section 5, as well as the Isabelle theories `Identifiers` and `SML_Syntax` which describe the syntax of the language $\mathcal{F}$.

The theory hierarchy is shown in the Figure 2 (the existing ZF theories are shown boxed and the new theories defined for the **Elle** system are shown unboxed).

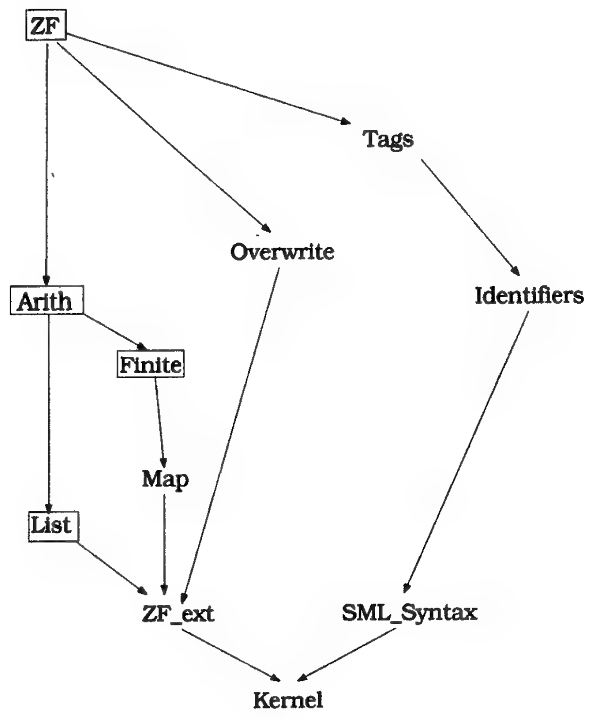

Figure 2 Kernel Theory Hierarchy

8.1 The Theory `Identifiers`

The theory `Identifiers` is an extension of `Tags`, and defines the various classes of identifiers used in SML, as well as defining a number of built-ins and reserved names for each of these classes — all guaranteed to be distinct by virtue of the machinery established in `Tags`.

Table 6 shows the various classes of identifiers (each is a set), and lists the associated built-ins and reserved names. For example, *plus* is a built-in function, whose meaning for evaluation is recorded in the initial (dynamic) environment in terms of the semantic *apply* function [1], and whose type is recorded in the initial static environment. The variables $x_1, ..., x_9$ are guaranteed via the `Tags` mechanism to be distinct, and can be used for general-purpose value variables; we also have $f_1, ..., f_9$, which can be used for user-defined ML functions. The variable *it* is special: it is used to record the value of the most recently entered expression. The record labels n_1 and n_2 are special ones which denote the components of an ordered pair, while $p_1, ..., p_6$ can be used for general record labels.

Table 6 Identifiers Used in the Kernel

Name of Class	Description	Built-Ins	Reserved Names
Var	value variables	$plus, minus, times, less$ $zero, not,$ $head, tail, null$	$x_1, \ldots, x_9$ $f_1, \ldots, f_9$ it
Con	value constructors	$true, false, nil, CONS$	$c_1, \ldots, c_9$
ExCon	exception constructors	$ERROR, MATCH, BIND$	
TyVar	type variables		$a_1, \ldots, a_9$
TyCon	type constructors	$INT, BOOL, UNIT, EXN, List$	$T_1, \ldots, T_9$
Lab	record labels		n_1, n_2 $p_1, \ldots, p_6$
StrId	structure identifiers		$S_1, \ldots, S_9$

8.2 The Theory `SML_Syntax`

Our aim has been to use wherever possible the syntax of SML. Table 7 gives the concrete syntax for the bare (i.e. nonderived) part of $\mathcal{F}$, the subset of SML which is supported by the **Elle** system. We adopt the conventions of the Definition [1]: in particular, we have used the angled brackets '< >' to enclose optional phrases.

Table 7 Syntax of Bare Part of $\mathcal{F}$

exp :: =	scon *n*	*pat* :: =	_
	var *x*		scon *n*
	con *c*		var *x*
	excon *e*		con *c*
	{{ <*exprow*> }}		excon *e*
	let *dec* in *exp* end		{{ <*patrow*> }}
	(- *exp* -)		(- *exp* -)
	exp_1 ` exp_2		con con *c* *pat*
	exp_1 *id* exp_2		excon *e* *pat*
	exp : *ty*		pat_1 con *c* pat_2
	exp handle *match*		pat_1 excon *e* pat_2
	raise *exp*		var *x* <: *ty*> as *pat*
	fn *match*		*pat* : *ty*
exprow :: =	*lab* = *exp* < ,*exprow*>	*patrow* :: =	...
match :: =	*pat* => *exp* < | *match*>		*pat* = *exp* < ,*patrow*>
dec :: =	val *valbind*	*ty* :: =	*tyvar*
	type *typbind*		{{ <*tyrow*>}}
	datatype *datbind*		*tyseq* *tycon*
	abstype *datbind* with *dec* end		*ty* -ty-> *ty*'
	exception *exbind*		(- *ty* -)
	localdec_1 in dec_2 end	*tyrow* :: =	*lab* : *ty* < , *tyrow*>
	dec_1 <;> dec_2		
valbind	*pat* = *exp* <and *valbind*>		
	rec *valbind*		
typbind	*tyvarseq* *tycon* = *ty* <and *typbind*>		
datbind	*tyvarseq* *tycon* is *conbind* <and *datbind*>		
conbind	con *c* <of *ty*> <| *conbind*>		
exbind	excon *e* <of *ty*> <and *exbind*>		
	excon *e1* = excon *e2* <and *exbind*>		

For example, the factorial function is given by

```
val rec var f1 = fn var x1 =>
   (- if zero var x1 then scon succ (0) else
   (- var x1 ** (- var f1 ` (- var x1 -- scon succ (0) -)-)-)-)
```

Some of the relevant productions are

$$exp \rightarrow \mathtt{fn}\ match$$
$$match \rightarrow pat \Rightarrow exp\ \langle\ |\ match\rangle$$

which are easily expressed as the following Isabelle syntax declarations:

```
Fn      ::  "Match => Exp"                    ("(1fn _)")
Mrule   ::  "[ Pat, Exp] =>  Match"           ("(_ =>/ _)")
Match_  ::  "[ Pat, Exp, Match] =>  Match"  ("(_ =>/ _ |/ _)")
```

Note that `Match` and `Exp` appear as new types in the logic; thus Isabelle's type-checking will catch syntax errors.

The definition of the syntax is taken almost verbatim from the Definition, with the following important differences.

- A number of constructs need some extra concrete syntax as "dressing" to enable Isabelle to disambiguate them from other constructs. This can be seen in the factorial example above, in which the variables are marked with the word `var`. Other examples are special constants, value constructors and exception constructors, as well as the analogous patterns and constructor bindings.
- The phrase classes of atomic expressions (*AtExp*) and expressions (*Exp*) are not distinguished, and have been collapsed here into the single class *Exp*. The same applies to atomic patterns (*AtPat*) and patterns (*Pat*). Also the phrase class *Mrule* is not used.
- Isabelle allows some limited overloading — for example, record expressions and patterns are both denoted by the same double brackets, such as `{{p1 = var x1, p2 = var x2}}`. Unfortunately, use of single brackets would lead to confusion with ZF sets!
- Brackets in the language are denoted by `(- ... -)`, to avoid confusion with Isabelle's meta-level brackets `( ... )`
- Function application is denoted by a backquote, such as `var f1 ` var x1`. Simple juxtaposition, as in `var f1 var x1`, gets confused with Isabelle's meta-application.
- For type bindings, the syntax is `tyvarseq tycon is ty`, and not `tyvarseq tycon = ty` , which causes Isabelle problems in parsing
- Function types are denoted by `A -ty-> B`
- Open declarations, fixity directives and 'op' qualifiers have been ignored.

In addition to the above bare constructs, SML has a large number of derived forms. Table 8 shows the derived forms supported by the **Elle** system.

Table 8 Syntax of Derived Forms of $\mathcal{F}$

exp ::=	`(-` exp_1`,`exp_2`-)`	*pat* ::=	`(-` pat_1`,`pat_2`-)`
	exp_1 `+` exp_2	*ty* ::=	ty_1 `<*>`ty_2
	exp_1 `-` exp_2		`Int`
	exp_1 `*` exp_2		`Bool`
	exp_1 `<` exp_2		`Exn`
	exp_1 `orelse` exp_2	*tyseq* ::=	`<>`
	exp_1 `andalso` exp_2		<*tyargs*>
	`if` exp_1 `then` exp_2 `else` exp_3	*program* ::=	*exp* `;`
	`case` *exp* `of` *match*	*expseq* ::=	*exp*
	`zero` *exp*		*exp* `,` *expseq*
	`not` *exp*	*tyargs* ::=	*ty*
	`head` *exp*		*ty* `,` *tyargs*
	`tail` *exp*		
	`null` *exp*		
	`[]`		
	[*expseq*]		
	exp_1 `::` exp_2		

9 Evaluation (Dynamic Semantics)

In this section we shall describe how the dynamic semantics of the language $\mathcal{F}$ is captured within Isabelle. We shall follow the treatment given in Chapter 6 of the Definition [1]. This reference should be consulted for more detailed explanations of the terms used.

9.1 The Theory `EvalObj`

The semantic objects for evaluation are all defined within the theory `EvalObj`, which is an extension of `Kernel`.

The simple semantic objects are as follows:

Addr	addresses
ExName	exception names
BasVal	basic values
SVal	special values
FAIL	failure element

$Addr$ and $ExName$ are infinite sets. $BasVal$ denotes the set of values bound to predefined variables, i.e. $BasVal = \{plus, minus, times, less, zero, not, head, tail, null\}$. $SVal$ is the set of values denoted by the special constants $SCon$. FAIL is used in the semantics simply to record the failure of a pattern to match a value. All these sets are declared in `EvalObj`.

The compound semantic objects ([1], p47) are defined as follows

$$
\begin{aligned}
Val &\equiv SVal \sqcup BasVal \sqcup Con \sqcup ConVal \sqcup \\
&\qquad ExVal \sqcup Record \sqcup Addr \sqcup Closure \\
Record &\equiv Lab \overset{fin}{\to} Val \\
ExVal &\equiv ExName \sqcup (ExName \times Val) \\
Pack &\equiv \{Raised\} \times ExVal \\
Conval &\equiv Con \times Val \\
Mem &\equiv Addr \overset{fin}{\to} Val \\
ExNameSet &\equiv Fin(ExName) \\
State &\equiv Mem \times ExNameSet \\
Env &\equiv StrEnv \times VarEnv \times ExConEnv \\
StrEnv &\equiv StrId \overset{fin}{\to} Env \\
VarEnv &\equiv (Var \sqcup Con \sqcup ExCon) \overset{fin}{\to} Val \\
ExConEnv &\equiv ExCon \overset{fin}{\to} ExName
\end{aligned}
$$

On comparison, it should be clear how the use of ZF has allowed us to follow the Definition extremely closely (note, however, that we have omitted the `:=` operator from Val).

Environments (i.e. members of Env) are combined by

$$\langle SE, VE, EE\rangle \dagger \langle SE', VE', EE'\rangle \equiv \langle SE \dagger SE', VE \dagger VE', EE \dagger EE'\rangle$$

where the angle brackets are standard ZF notation for tuples.

The initial environment is given by the rules

$$
\begin{aligned}
&E_0 \equiv \langle SE_0, VE_0, EE_0\rangle \\
&E_0 \in Env
\end{aligned}
$$

where

$$\begin{aligned}
&SE_0 \equiv \emptyset \\
&VE_0 \equiv \{\langle\langle Var, plus\rangle\langle BasVal, plus\rangle\rangle, \\
&\qquad \langle\langle Var, minus\rangle\langle BasVal, minus\rangle\rangle, \\
&\qquad \langle\langle Var, times\rangle, \langle BasVal, times\rangle\rangle, \\
&\qquad \langle\langle Var, less\rangle, \langle BasVal, less\rangle\rangle, \\
&\qquad \langle\langle Var, zero\rangle, \langle BasVal, zero\rangle\rangle, \\
&\qquad \langle\langle Var, not\rangle, \langle BasVal, not\rangle\rangle, \\
&\qquad \langle\langle Var, head\rangle, \langle BasVal, head\rangle\rangle, \\
&\qquad \langle\langle Var, tail\rangle, \langle BasVal, tail\rangle\rangle, \\
&\qquad \langle\langle Var, null\rangle, \langle BasVal, null\rangle\rangle, \\
&\qquad \langle\langle Con, true\rangle, \langle Con, true\rangle\rangle, \\
&\qquad \langle\langle Con, false\rangle, \langle Con, false\rangle\rangle, \\
&\qquad \langle\langle Con, nil\rangle, \langle Con, nil\rangle\rangle, \\
&\qquad \langle\langle Con, CONS\rangle, \langle Con, CONS\rangle\rangle\} \\
&EE_0 \equiv \emptyset
\end{aligned}$$

The meaning of built-ins such as *plus* is given by the semantic *apply* function. For example, the rule for the built-in *plus* is most usefully expressed as folllows

$$\begin{gathered}(\exists k.m + n = k \;\&\; \langle SVal, k\rangle = v) \Leftrightarrow \\ apply(\langle BasVal, plus\rangle.(m, n)) = v\end{gathered}$$

where the value (m, n) is a kind of derived form, and is a shorthand for

$$\langle\langle Record, \langle n_1, SVal, m\rangle, \langle n_2, SVal, n\rangle\rangle\rangle$$

The rules for the other built-ins are rather lengthy, and will not be given here.

The dynamic semantics needs to have *function closures*: in order to achieve correct call-time environments, the apply function is extended to closures. When we apply $closure(match, E, VE)$ to the value v, $match$ is evaluated against v, in the environment E modifed by *Rec VE*. For details of these technical issues, we refer to the Definition (p.49); it suffices to say here that they are handled correctly by the **Elle** system.

Reasoning about semantic objects can be carried out by the following tactics:

```
obj_step_tac : int -> tactic
apply_tac : int -> tactic
obj_tac : int -> tactic
```

Essentially, `obj_step_tac` carries out a single resolution step using appropriate introduction rules derived from `EvalObj`. The tactic `apply_tac` takes care of goals involving the built-in functions plus, minus etc. Finally, `obj_tac` uses depth-first search to solve a particular subgoal.

```
val obj_tac = DEPTH_SOLVE_1 o obj_step_tac;
```

9.2 The Theories `EvalAtExp, ..., EvalProgram`

9.2.1 Inference Rules

For the most part, the inference rules capturing the dynamic semantics of $\mathcal{F}$ are easily expressed in Isabelle.

The theories `EvalAtExp`, ... , `EvalProgram` form a linear chain extending `EvalObj`. Below, we give the detailed rules for the theory `EvalAtExp` (these correspond to rules 103 to 109 in the Definition).

$$\frac{\langle \mathit{Val}, \mathit{SVal}, n\rangle = v'}{E \vdash \texttt{scon}\ n \Rightarrow v'} \quad (\text{Scon} \Updownarrow)$$

$$\frac{(\exists SE\ VE\ EE.E = \langle SE, VE, EE\rangle \wedge VE(\langle \mathit{Var}, x\rangle) = v \wedge \langle \mathit{Val}, v\rangle = v')}{E \vdash \texttt{var}\ x \Rightarrow v'} \quad (\text{Var} \Updownarrow)$$

$$\frac{\langle \mathit{Val}, \mathit{Con}, c\rangle = v'}{E \vdash \texttt{con}\ c \Rightarrow v'} \quad (\text{Con} \Updownarrow)$$

$$\frac{\begin{array}{c}(\exists SE\ VE\ EE.E = \langle SE, VE, EE\rangle \wedge EE(\langle \mathit{ExCon}, e\rangle) = en\ \wedge \\ \langle \mathit{Val}, \mathit{ExVal}, \mathit{ExName}, en\rangle = v')\end{array}}{E \vdash \texttt{excon}\ e \Rightarrow v'} \quad (\text{ExCon} \Updownarrow)$$

$$\frac{\langle \mathit{Val}, \mathit{Record}, \emptyset\rangle = v'}{E \vdash \{\{\ \}\} \Rightarrow v'} \quad (\text{EmptyRecord} \Updownarrow)$$

$$\frac{(\exists record.E \vdash exprow \Rightarrow \langle \mathit{Val}, record\rangle \wedge \langle \mathit{Val}, \mathit{Record}, record\rangle = v')}{E \vdash \{\{exprow\}\} \Rightarrow v'} \quad (\text{Record} \Updownarrow)$$

$$\frac{(\exists E'\ E''.E \vdash dec \Rightarrow E' \wedge E \dagger E' = E'' \wedge E'' \vdash exp \Rightarrow v')}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v'} \quad (\text{Let} \Updownarrow)$$

$$\frac{E \vdash exp \Rightarrow v'}{E \vdash (-exp-) \Rightarrow v'} \quad (\text{Bracket} \Updownarrow)$$

The following rule is important:

$$\frac{(\exists\ VE.E \vdash valbind \Rightarrow\ VE \wedge \langle 0, VE, 0\rangle = E')}{E \vdash \texttt{val}\ valbind\ \Rightarrow E'} \quad (\mathit{Val} \Updownarrow)$$

These rules are all two-way (symmetric) rules involving logical equivalence between the conclusion and the conjunction of the premises: for convenience, such rules are identified by the symbol $\Updownarrow$ in their name.

9.2.2 Introduction Rules

The first stage is to derive the necessary introduction rules for sequents involving all the possible kinds of language phrase. (We refer to the general discussion in Section 4).

As a first example, consider the rule Scon $\Updownarrow$. The naive introduction rule would be

$$\frac{\langle \mathit{Val}, \mathit{SVal}, n \rangle = v'}{E \vdash \texttt{scon}\ n \Rightarrow v'} \quad \text{(SconI)}$$

However, it is more useful to make the equality substitution for v', and use the rule in the form

$$E \vdash \texttt{scon}\ n \Rightarrow \langle \mathit{Val}, \mathit{SVal}, n \rangle \quad \text{(SconI)}$$

As a second (contrasting) example, consider the rule Let $\Updownarrow$. In this case, the obvious introduction rule

$$\frac{E \vdash \mathit{dec} \Rightarrow E'; E \dagger E' = E'' \quad E'' \vdash \mathit{exp} \Rightarrow v'}{E \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow v'} \quad \text{(LetI)}$$

is the one which must be used. We might be tempted to make the equality substitution for E'', giving

$$\frac{E \vdash \mathit{dec} \Rightarrow E' \quad E \dagger E' \vdash \mathit{exp} \Rightarrow v'}{E \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow v'} \quad \text{(LetI)}$$

However, this form of the rule would make it difficult to establish the value of $E \dagger E'$ (with E known, and E' unknown), and thus the Isabelle proof would falter.

In general, we make equality substitutions in deriving introduction rules only when the term appearing after the substitution is made does not need to be subjected to further reasoning to obtain its value.

The appropriate introduction rule for *Val* is

$$\frac{E \vdash \ \mathit{valbind} \Rightarrow \mathit{VE}}{E \vdash \texttt{val}\ \mathit{valbind} \Rightarrow \langle \mathit{0}, \mathit{VE}, \mathit{0} \rangle} \quad \text{(ValI)}$$

We use the ML variable `intr_rules` to denote the list of all introduction rules for the evaluation of language sequents. These are supplemented by the special rules

$$\frac{\langle \langle \mathit{Var}, x \rangle, v \rangle \in \mathit{VE_0}}{E_0 \vdash \texttt{var}\ x \Rightarrow v} \quad \text{(Var\_init)}$$

$$\langle \langle \mathit{Var}, \mathit{plus} \rangle, \mathit{BasVal}, \mathit{plus} \rangle \in \mathit{VE_0} \qquad \text{(VE0\_mems)}$$

$$\cdots$$

$$\langle \langle \mathit{Con}, \mathit{CONS} \rangle, \mathit{Con}, \mathit{CONS} \rangle \in \mathit{VE_0}$$

which hide some of the tedious reasoning about built-ins.

9.3 Proof Procedures

The aim of the Isabelle proof procedures for evaluation is to support reasoning which enables one to establish the truth of sequents of the form

$$E \vdash phrase \Rightarrow result$$

in which E is a (fixed) environment, and the language phrase is typically an expression, declaration or program. The result of the evaluation may or may not be known — if not, it is modelled by an Isabelle scheme variable to be instantiated during the proof.

The proof procedures are simple Isabelle tactics which capture the way one would construct a proof tree [29]. Starting from the initial goal (the root of the tree), we keep resolving with the appropriate inference rules, simplifying environments as we go, until all language phrases have disappeared. We then simplify using the inference rules for semantic objects until we reach the leaves of the tree. Isabelle keeps track of the instantiations made as we go. Evaluations which involve pattern matching may require backtracking search. To accommodate this, our proof procedures use the Isabelle depth-first search strategy.

The basic Isabelle step tactics for evaluation are:

```
eval_step_tac : int -> tactic
step_tac : int -> tactic
```

The aim of `eval_step_tac` is to apply one of the introduction rules for sequents, in a controlled fashion. The sequent $E \vdash phrase \Rightarrow result$ is called *indeterminate* if $phrase$ is a scheme variable. Clearly, in the usual case where $result$ is a scheme variable, an indeterminate sequent can be resolved with any introduction rule, leading to erroneous paths in the proof tree; we wish to avoid this by making `eval_step_tac` check that the subgoal does not contain an indeterminate sequent, and fail if it does. To do this, we use the function

```
SUBGOAL : (term * int -> tactic) -> int -> tactic
```

which allows a tactic to inspect a particular subgoal, and take appropriate action.

The all-purpose `step_tac` tries in turn to reduces a sequent, if possible; to use the introduction rules for the semantic apply function; otherwise it attacks the subgoal with `obj_tac`.

```
val step_tac = eval_step_tac ORELSE'
                   apply_tac ORELSE' obj_tac;
```

The higher-level tactic `evaluate_tac` is the iterative form of `step_tac`. The most useful tactic is `eval_tac`: this carries out a depth-first search.

```
val evaluate_tac  = REPEAT1 (step_tac 1);
val eval_tac  = DEPTH_SOLVE evaluate_tac;
```

10 Elaboration (Static Semantics)

The support for reasoning about the static semantics of the language $\mathcal{F}$ is set up along very similar lines to that for evaluation. Again, we refer the reader to the Definition [1] for further information.

10.1 The Theory `ElabObj`

The semantic objects for elaboration are all defined within the theory `ElabObj`, which is an extension of `Kernel`.

The simple semantic objects are as follows: $TyName$ and $StrName$ are infinite sets,

TyVar	type variables
TyName	type names
StrName	structure names

declared in `ElabObj`. The class $TyVar$, already defined in `Kernel`, is also extensively used.

The compound semantic objects ([1], p. 17) are defined as follows

$$
\begin{aligned}
&Type \equiv TyVar \sqcup RecType \sqcup FunType \sqcup ConsType\\
&RecType \equiv Lab \overset{fin}{\rightarrow} Type\\
&Funtype \equiv Type \times Type\\
&ConsType \equiv list(Type) \times TyName\\
&TypeFcn \equiv \{Lam\} \times list(TyVar) \times Type\\
&TypeScheme \equiv \{All\} \times Pow(TyVar) \times Type\\
&Str \equiv StrName \times Env\\
&TyStr \equiv TypeFcn \times ConEnv\\
&StrEnv \equiv StrId \overset{fin}{\rightarrow} Str\\
&TyEnv \equiv TyCon \overset{fin}{\rightarrow} TyStr\\
&ConEnv \equiv Con \overset{fin}{\rightarrow} TypeScheme\\
&VarEnv \equiv (Var \sqcup Con \sqcup ExCon) \overset{fin}{\rightarrow} TypeScheme\\
&ExConEnv \equiv ExCon \overset{fin}{\rightarrow} Type\\
&Env \equiv StrEnv \times TyEnv \times VarEnv \times ExConEnv\\
&TyNameSet \equiv Fin(TyName)\\
&TyVarSet \equiv Fin(TyVar)\\
&Context \equiv TyNameSet \times TyVarSet \times Env
\end{aligned}
$$

Again, as for evaluation, we have been able to follow the Definition extremely closely. Note the way that the compound objects $TypeFcn$ and $TypeScheme$ are defined, using the singleton sets $\{Lam\}$ and $\{All\}$ as distinguishing labels.

Contexts and environments are combined as follows

$$\langle T, U, E\rangle \dagger \langle T', U', E'\rangle \equiv \langle T \cup T', U \cup U', E \dagger E'\rangle$$

$$\langle SE, TE, VE, EE\rangle \dagger \langle SE', TE', VE', EE'\rangle \equiv \langle SE \dagger SE', TE \dagger TE', VE \dagger VE', EE \dagger EE'\rangle$$

There are a number of predefined quantities:

$$\begin{gathered}
int \equiv \langle ConsType, \emptyset, INT\_NAME\rangle \\
bool \equiv \langle ConsType, \emptyset, BOOL\_NAME\rangle \\
exn \equiv \langle \mathrm{ConsType}, \emptyset, \mathrm{EXN\_NAME}\rangle \\
unit \equiv \langle ConsType, \emptyset, UNIT\_NAME\rangle \\
t_1[*]t_2 \equiv \langle RecType, \{\langle n_1, t_1\rangle, \langle n_2, t_2\rangle\}\rangle \\
INT\_NAME \in TyName \\
BOOL\_NAME \in TyName \\
EXN\_NAME \in TyName \\
UNITNAME \in TyName
\end{gathered}$$

The initial context is given by the rules

$$\begin{aligned}
&C_0 \equiv \langle T_0, U_0, SE_0, TE_0, VE_0, EE_0\rangle \\
&C_0 \in Context, \\
&\text{where}: \\
&T_0 \equiv \emptyset \\
&U_0 \equiv \emptyset \\
&SE_0 \equiv \emptyset \\
&TE_0 \equiv \{\langle\langle TyCon, UNIT\rangle, \langle Lam, \emptyset, RecType, \emptyset\rangle, \emptyset\rangle \\
&\qquad\quad \langle\langle TyCon, BOOL\rangle, \langle Lam, \emptyset, bool\rangle, \\
&\qquad\qquad\quad \{\langle\langle Con, true\rangle, bool\rangle, \langle\langle Con, false\rangle, bool\rangle\}\rangle, \\
&\qquad\quad \langle\langle TyCon, INT\rangle, \langle Lam, \emptyset, int\rangle, \emptyset\rangle, \\
&\qquad\quad \langle\langle TyCon, List\rangle, \\
&\qquad\qquad \langle Lam, \{alpha\}, ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, LIST\_NAME\rangle, \\
&\qquad\qquad \{\langle\langle Con, nil\rangle, \langle All, \{alpha\}, \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle\rangle\rangle, \\
&\qquad\qquad\; \langle\langle Con, CONS\rangle, \langle All, \{alpha\}, \langle FunType, \langle\langle TyVar, alpha\rangle[*],\rangle\rangle\rangle\rangle\} \\
&\qquad\qquad\quad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle, \\
&\qquad\qquad\qquad\qquad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle\rangle\rangle\rangle\}\rangle, \\
&\qquad \langle\langle TyCon, EXN\rangle, \langle Lam, \emptyset, exn\rangle, \emptyset\rangle\}
\end{aligned}$$

$$
\begin{aligned}
VE_0 \equiv \{ & \langle\langle Var, plus\rangle, \langle All, 0, FunType, int[*]int, int\rangle\rangle, \\
& \langle\langle Var, minus\rangle, \langle All, 0, FunType, int[*]int, int\rangle\rangle, \\
& \langle\langle Var, times\rangle, \langle All, 0, FunType, int[*]int, int\rangle\rangle, \\
& \langle\langle Var, zero\rangle, \langle All, 0, FunType, int, bool\rangle\rangle, \\
& \langle\langle Var, not\rangle, \langle All, 0, FunType, bool, bool\rangle\rangle, \\
& \langle\langle Var, head\rangle, \langle All, \{alpha\}, FunType, \\
& \quad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle, \langle TyVar, alpha\rangle\rangle\rangle, \\
& \langle\langle Var, tail\rangle, \langle All, \{alpha\}, FunType, \\
& \quad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle, \\
& \quad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle\rangle\rangle, \\
& \langle\langle Var, null\rangle, \langle All, \{alpha\}, FunType, \\
& \quad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle, bool\rangle\rangle, \\
& \langle\langle Con, true\rangle, \langle All, 0, bool\rangle\rangle, \\
& \langle\langle Con, false\rangle, \langle All, 0, bool\rangle\rangle, \\
& \langle\langle Con, nil\rangle, \langle All, \{alpha\}\langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle\rangle\rangle, \\
& \langle\langle Con, CONS\rangle, \langle All, \{alpha\}, \langle FunType, \langle TyVar, alpha\rangle[*], \\
& \qquad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle, \\
& \qquad \langle ConsType, \langle\langle TyVar, alpha\rangle, 0\rangle, List\rangle\rangle\rangle\rangle\}
\end{aligned}
$$

$$EE0 \equiv \emptyset$$

Polymorphic type inference is an important and difficult aspect of the elaboration of SML phrases. A considerable amount of machinery is needed to allow sound polymorphic typing.

Type schemes, i.e. types quantified over type variables, are used to capture polymorphism. For example, the elaboration of the function Id defined by

```
    val Id = fn x => x
```

results in the type environment

$$\{\langle Id, \forall\alpha.\alpha \to \alpha\rangle\}$$

In general, if σ is a type scheme is of the form:

$$\sigma = \forall\alpha_1 ... \alpha_n.\tau$$

then τ' is an *instance* of σ, written $\sigma \succ \tau'$ if

$$\tau' = \tau\{\tau_1/\alpha_1, ..., \tau_n/\alpha_n\}$$

Different occurrences of a variable may have different instances of the same type scheme. Thus, in `Id (Id)`, the first occurrence of `Id` has the type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ and the second has type $\alpha \to \alpha$. Each is an instance of $\forall\alpha.\alpha \to \alpha$

In order to preserve the soundness of the type inference system for SML, the elaboration of *ValBind* must make use of *closure* operations in order that type variables can be used polymorphically (see Rule 17 in the Definition [1]). Note that this closure is not to be confused with the function closures needed for successful evaluation of function expressions at call time.

If τ is a type, and A a semantic object, we define

$$close_A(\tau) = \forall\alpha^{(k)}.\tau \quad \text{where} \quad \alpha^{(k)} = tyvars(\tau)\backslash tyvars(A)$$

where $tyvars(A)$ denotes the set of type variables which are free in A. If the range of a variable environment VE contains only types, we set

$$Close_A(VE) = \{\langle id, close_A(\tau)\rangle, VE(id) = \tau\}$$

(this is naturally extended to $Close_A(E)$, where E is an environment).

How is all this represented in ZF? We represent the type scheme

$$\forall\alpha_1 \ldots \alpha_n.\tau$$

as the ZF tuple

$$\langle All, \{\alpha_1, \ldots \alpha_n\}, \tau\rangle$$

The following rules specify when a type can be an instance of a type scheme.

$$\langle All, \emptyset, \tau\rangle \succ \tau$$

$$\frac{\langle All, A, \tau(\tau')\rangle \succ u}{\langle All, cons(\alpha, A), \tau(\langle TyVar, \alpha\rangle)\rangle \succ u}$$

The following rules define $tyvars(A)$:

$$\begin{aligned}
&tyvars\langle TyVar, \alpha\rangle \equiv \{\alpha\}\\
&tyvars\langle FunType, \tau, \tau'\rangle \equiv tyvars\ \tau \cup tyvars\ \tau'\\
&tyvars\langle RecType, \emptyset\rangle \equiv \emptyset\\
&tyvars\langle RecType, cons(\langle label, \tau\rangle, r)\rangle\\
&\quad \equiv tyvars\ \tau \cup tyvars\ \langle RecType, r\rangle\\
&tyvars\langle ConsType, \emptyset, tyname\rangle \equiv \emptyset\\
&tyvars\langle ConsType, \langle\tau, \tau'\rangle\rangle\\
&\quad \equiv tyvars\ \tau \cup tyvars\ \langle ConsType, \tau', tyname\rangle
\end{aligned}$$

For type schemes and type functions, we have

$$tyvars\langle All, A, \tau\rangle \equiv tyvars\ (\tau) \setminus A$$

$$tyvars\langle\langle Lam, A, \tau\rangle, CE\rangle \equiv tyvars\ (\tau) \cup tyvars\ CE \setminus A$$

For environments and contexts the rules are

$$\begin{aligned}
&Tyvars\ \emptyset \equiv \emptyset\\
&Tyvars\ cons(\langle\langle Index, x\rangle, \tau\rangle, VE) \equiv\\
&\quad tyvars\ \tau \cup Tyvars\ VE\\
&Tyvars\ cons(\langle\langle TyCon, tycon\rangle, \theta, CE\rangle, TE) \equiv\\
&\quad tyvars\ \theta \cup Tyvars\ C\ E \cup Tyvars\ TE\\
&Tyvars\langle c, cs\rangle \equiv Tyvars\ c \cup Tyvars\ cs
\end{aligned}$$

Now we can compute closures via

$$\begin{aligned}
&close(A, \langle All, \emptyset, \tau\rangle) \equiv \langle All, tyvars\ \tau \setminus Tyvars\ A, \tau\rangle\\
&Close(A, \emptyset) \equiv \emptyset\\
&Close(A, cons(\langle ide, \tau\rangle, VE)) \equiv\\
&\quad cons(\langle ide, close(A, \tau)\rangle, Close(A, VE))
\end{aligned}$$

We also need a rule for carrying out substitutions. The formula $subst\ (\tau[\tau'/\alpha], \tau'')$ is true when τ'' is the result of substituting τ' for the type variable α in the type τ. The rule is conveniently obtained by use of Isabelle's meta-level functions in the form

$$subst\ \big(\tau(\langle TyVar, \alpha\rangle)[\tau'/a], \tau(\tau')\big)$$

Although this is very simple formulation, and has always worked in our examples, it has the disadvantage of requiring that the meta-level substitution works correctly for occurrences of the subexpression $\langle TyVar,\alpha\rangle$. An alternate way of specifying $subst$ would be to provide explicit recursion equations.

Reasoning about semantic objects is carried out by the following ML functions:

```
obj_step_tac : int -> tactic
obj_tac : int -> tactic
```

The code will not be given here. However, we note that, if `obj_step_tac` is unable to make progress, then it replaces each scheme variable representing a type variable name by a constant. This is done throughout the proof state, by means of the function `set_tyvars : thm -> thm`. If in a given theorem `th` there are n such 'unknown' type variables of the form $\langle TyVar, ?v\rangle$, then `set_tyvars th` is the theorem which results from replacing these by the distinct type variables $\langle TyVar, a_1\rangle, ..., \langle TyVar, a_n\rangle$. This is done to capture ML's polymorphic type inference within our Isabelle theory. An example is given later in the paper.

Finally, `obj_tac` uses depth-first search to solve a particular subgoal.

```
val obj_tac = DEPTH_SOLVE_1 o obj_step_tac;
```

10.2 The Theories `ElabAtExp`, ..., `ElabProgram`

10.2.1 Inference Rules

As with the dynamic semantics, the static semantics inference rules for $\mathcal{F}$ are easily given in Isabelle.

The theories `ElabExp`, ... , `ElabProgram` form a linear chain extending `ElabObj`. Below, we give the detailed rules for the theory `ElabExp` (Rules 1–7 in the Definition) — compare these with the corresponding evaluation rules from `EvalExp` given above in 9.2.

$$\frac{int = t}{C \vdash \texttt{scon}\ n \Rightarrow t}\ (\text{Scon} \Updownarrow)$$

$$\frac{\begin{array}{c}(\exists T\ U\ SE\ VE\ EE.C = \langle T, U, SE, TE, VE, EE\rangle \wedge \\ VE`\langle Var, x\rangle = s \wedge s \succ t)\end{array}}{C \vdash \texttt{var}\ x \Rightarrow t}\ (\text{Var} \Updownarrow)$$

$$\frac{\begin{array}{c}(\exists T\ U\ SE\ VE\ EE.C = \langle T, U, SE, TE, VE, EE\rangle \wedge \\ VE`\langle Con, c\rangle = s \wedge s \succ t)\end{array}}{C \vdash \texttt{con}\ c \Rightarrow t}\ (\text{Con} \Updownarrow)$$

$$\frac{\begin{array}{c}(\exists T\ U\ SE\ VE\ EE.C = \langle T, U, SE, TE, VE, EE\rangle \wedge \\ VE`\langle ExCon, c\rangle = s \wedge s \succ t)\end{array}}{C \vdash \texttt{excon}\ c \Rightarrow t}\ (\text{ExCon} \Updownarrow)$$

$$\frac{\langle RecType, \emptyset\rangle = t}{C \vdash \{\{\}\} \Rightarrow t}\ (\text{EmptyRecord} \Updownarrow)$$

$$\frac{(\exists rectype.C \vdash exprow \Rightarrow rectype \wedge \langle Rectype, rectype\rangle = t)}{C \vdash \{\{exprow\}\} \Rightarrow t}\ (\text{Record} \Updownarrow)$$

$$\frac{(\exists E\ C'.C \vdash dec \Rightarrow E \wedge C \dagger \langle 0, 0, E\rangle = C' \wedge C' \vdash exp \Rightarrow t)}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow t}\ (\text{Let} \Updownarrow)$$

$$\frac{C \vdash exp \Rightarrow t}{C \vdash (-exp-) \Rightarrow t}\ (\text{Bracket} \Updownarrow)$$

The following rule is important for type inference:

$$\frac{\begin{array}{c}(\exists\ VE\ VE'\ VE''.C \vdash valbind \Rightarrow VE \wedge VE = VE' \\ \wedge Close_C(VE') = VE'' \wedge \langle 0, 0, VE'', 0\rangle = E)\end{array}}{C \vdash \texttt{val}\ valbind \Rightarrow E}\ (\text{Val} \Updownarrow)$$

10.2.2 Introduction Rules

As with evaluation, we need to make certain equality substitutions in the two-way rules capturing the static semantics. Again, `intr_rules` denotes the list of all introduction rules for the evaluation of language sequents. These are supplemented by the following special rules

$$\frac{\langle\langle Var, x\rangle, v\rangle \in VE_0 \quad s \succ t}{C_0 \vdash \texttt{var}\ x \Rightarrow t}\ (\text{Var\_init})$$

...

$$\langle\langle Var, plus\rangle, All, 0, FunType, \langle RecType, \{\langle n_1, int\rangle, \langle n_2, int\rangle\}\rangle, int\rangle \in VE_0$$

$$\dots$$
$$\langle\langle \mathit{TyCon}, \mathit{INT}\rangle, \langle \mathit{Lam}, \mathit{0}, \mathit{int}\rangle, \mathit{0}\rangle \in \mathit{TE}_0$$
$$\dots$$

Later on we shall need the introduction rules for *Let* and *Val*. They are as follows:

$$\frac{C \vdash \mathit{dec} \Rightarrow E \quad C \dagger \langle \mathit{0}, \mathit{0}, E\rangle = C' \quad C' \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \mathtt{let}\ \mathit{dec}\ \mathtt{in}\ \mathit{exp}\ \mathtt{end} \Rightarrow \tau} \quad (\mathrm{LetI})$$

$$\frac{C \vdash \ \mathit{valbind} \Rightarrow \mathit{VE} \quad \mathit{VE} = \mathit{VE}' \quad \mathit{Close}_C = \mathit{VE}''}{C \vdash \mathtt{val}\ \mathit{valbind} \Rightarrow \langle \mathit{0}, \mathit{0}, \mathit{VE}'', \mathit{0}\rangle} \quad (\mathrm{ValI})$$

10.3 Proof Procedures

The Isabelle proof procedures for elaboration are analogous to those for evaluation. Polymorphic type inference, as one would expect, requires special handling. However, the actual work of inferring the most general type of an expression is aided by Isabelle's scheme variables, which can then be instantiated to type variables.

The proof procedures for elaboration are aimed at establishing the truth of sequents of the form

$$C \vdash \mathit{phrase} \Rightarrow \mathit{result}$$

in which C is a (fixed) context; *phrase* is an expression, declaration or program; and *result* may be unknown.

The basic Isabelle step tactics for elaboration are:

```
elab_step_tac : int -> tactic
step_tac : int -> tactic
```

The aim of `elab_step_tac` is to apply one of the introduction rules for sequents, in a controlled fashion. The code is analogous to `eval_step_tac` and will not be given here. The code for `step_tac` is deceptively simple:

```
val step_tac = (FIRSTGOAL elab_step_tac) ORELSE obj_tac 1;
```

It tries to apply `elab_step_tac` to the first subgoal for which this tactic succeeds. If this is not possible, it attacks the subgoal 1 with `obj_tac`. This is a different approach from that used for evaluation, where subgoal 1 is the only one attacked. The reason for the difference is that the rule

$$\frac{(\exists C'.C \dagger \langle \mathit{0}, \mathit{0}, \mathit{0}, \mathit{0}, \mathit{VE}, \mathit{0}\rangle = C' \& C' \vdash \mathit{valbind} \Rightarrow \mathit{VE})}{C \vdash \mathtt{rec}\ \mathit{valbind} \Rightarrow \mathit{VE}} \quad (\mathrm{Rec} \Updownarrow)$$

actually allows us to infer the resulting variable environment VE in a recursive declaration! Therefore, some semantic goals must remain untouched until further instantiations have been made; hence the use of FIRSTGOAL.

As for evaluation, `elaborate_tac` is the iterative form of `step_tac`.

```
val elaborate_tac  = REPEAT1 (step_tac 1);
```

The tactic `elab_tac` itself does not require a depth-first search, but simply tidies up any floating unknown type variables.

```
        val elab_tac  = elaborate_tac THEN set_tyvars_tac;
```

where

```
        val set_tyvars_tac = Tactic
          (fn state => (marker := 1;
                        tapply (TRY (PRIMITIVE set_tyvars), state)));
```

11 Examples of Proofs

To illustrate our proof strategies, we shall show how our proof procedures handle the simple SML expression

```
        let val Id = fn x => x in Id end
```

The function Id has polymorphic type $\alpha \rightarrow \alpha$. We consider first the evaluation of this expression, which is straightforward, and then discuss its elaboration, which is more complex, owing to the need to compute closures of types and carry out the correct polymorphic type inference.

11.1 Evaluation of The Identity Function

First of all, we describe some of the proof steps which the tactic `eval_tac` uses to automate the evaluation of the identity function in the initial environment E_0.

We begin by setting the goal to be proved:

Level 0

$E_0 \vdash \mathtt{let}\ val\ Id = \mathtt{fn}\ x \Rightarrow x\ \mathtt{in}\ Id\ \mathtt{end} \Rightarrow ?v$

$1.\ E_0 \vdash \mathtt{let}\ val\ Id = \mathtt{fn}\ x \Rightarrow x\ \mathtt{in}\ Id\ \mathtt{end} \Rightarrow ?v$

Isabelle prints the level of the current proof state, with the numbered subgoals appeared underneath the goal to be proved. To improve clarity and readability, we shall present the output from Isabelle in a 'sanitised' way.[3]

Note that `eval_tac` would solve this goal immediately, but we shall work through some of the proof steps to illustrate how `eval_tac` works.

As described earlier, the first phase of the proof strategy is the repeated application of language inference rules, choosing the first subgoal containing a sequent, but avoiding those with indeterminate sequents.

The goal is of the form $E \vdash \mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \Rightarrow ?\mathtt{v}$. We therefore apply the introduction rule LetI from Section 9 to get three subgoals. The first subgoal calculates the environment arising from dec, the second overwrites the initial environment with the

[3] We have taken the liberty of dropping the `var` in expressions like `val var` $Id = \ldots$, cleaning up the names of some scheme variables generated during the proof, and using various (non-ASCII) Greek and mathematical symbols.

new one, and the last elaborates *exp* in the resulting environment:

Level 1

$E_0 \vdash$ `val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?v$

$1.\ E_0 \vdash$ `val` $Id =$ `fn` x `=>` $x\ \Rightarrow ?E'$

$2.\ E_0 \dagger\ ?E' = ?E''$

$3.\ ?E'' \vdash Id \Rightarrow ?v$

The first subgoal is a Val declaration, so we use ValI (see Section 9), to obtain

Level 2

$E_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?v$

$1.\ E_0 \vdash Id =$ `fn` x `=>` $x\ \Rightarrow ?VE$

$2.\ E_0 \dagger \langle 0, ?VE, 0\rangle = ?E''$

$3.\ ?E'' \vdash Id \Rightarrow ?v$

We continue with this process until no more language sequents remain:

Level 3

$E_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?v$

$1.\ (VE_0 \dagger \{\langle Id, closure(x \Rightarrow x, E_0, 0)\rangle\})(Id) = ?v$

Note the fact that the value recorded in the environment for the identity function is the function closure $closure(x \Rightarrow x, E_0, 0)$, which records information about the function necessary to evaluate calls to it correctly.

Making use of rules for applying overwrites, the proof is complete:

Level 4

$E_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow$

$\quad closure(x \Rightarrow x, E_0, 0)$

No subgoals!

The one-step proof of this using `eval_tac` took 0.4 seconds.[4]

11.2 Elaboration of The Identity Function

The elaboration of the identity function (in the initial context) is more involved. This time the goal is:

Level 0

$C_0 \vdash$ `let` *val* $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$

$1.\ C_0 \vdash$ `let` *val* $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$

The goal is of the form $C \vdash$ `let` *dec* `in` *exp* `end` $\Rightarrow ?\tau$. We therefore apply the introduction rule LetI (Section 10) to get three subgoals. The first subgoal calculates

[4] The timings given in this paper are for a Sparc 10 workstation with 64 MB of RAM.

the environment arising from *dec*, the second overwrites the initial context with this environment, and the last elaborates *exp* in the resulting context:

Level 1
$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$
1. $C_0 \vdash$ `val` $Id =$ `fn` x `=>` x $\Rightarrow ?E$
2. $C_0 \dagger \langle 0, 0, ?E\rangle = ?C'$
3. $?C' \vdash Id \Rightarrow ?\tau$

The first subgoal is a Val declaration, so we use ValI (Section 10). This rule decomposes the result of the declaration as a closure with respect to the initial context C_0. This results in three new subgoals (1–3 below). The subgoal involving an overwritten C_0 changes to a new subgoal (4) reflect the instantiation of $?E$ to the tuple $\langle 0, 0, ?VE'', 0\rangle$, and the third subgoal is carried over unchanged (as subgoal 5).

Level 2
$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$
1. $C_0 \vdash Id =$ `fn` x `=>` x $\Rightarrow ?VE$
2. $?VE = ?VE'$
3. $Close(C_0, ?VE') = ?VE''$
4. $C_0 \dagger \langle 0, 0, 0, 0, ?VE'', 0\rangle = ?C'$
5. $C' \vdash Id \Rightarrow ?\tau$

We continue with this process until no more language sequents remain:

Level 3
$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$
1. $C_0 \dagger \langle 0, 0, 0, 0, \{\langle x, All, 0, ?\tau'\rangle\}, 0\rangle = ?C_1'$
2. $?C_1' = \langle ?T, ?U, ?SE, ?TE, ?VE, ?EE\rangle$
3. $?VE(x) = ?\sigma$
4. $?\sigma \succ ?\tau''$
5. $\{\langle Id, All, 0, FunType, ?\tau', ?\tau''\rangle\} = ?VE'$
6. $Close(C_0, ?VE') = ?VE''$
7. $C_0 \dagger \langle 0, 0, 0, 0, ?VE'', 0\rangle = ?C'$
8. $?C' = \langle ?T', ?U', ?SE', ?TE', ?VE', ?EE'\rangle$
9. $?VE'(Id) = ?\sigma'$
10. $?\sigma' \succ ?\tau$

At this stage, we have removed all of the language reasoning, and all that is required is to prove the remaining semantic subgoals.

Expanding the context C_0, and simplifying gives

Level 4

$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$

1. $(VE_0 \dagger \{\langle x, All, 0, ?\tau'\rangle\}(x)) =?\sigma$
2. $?\sigma \succ ?\tau''$
3. $\{\langle Id, All, 0, FunType, ?\tau', ?\tau''\rangle\} =?VE'$
4. $Close(C_0, ?VE') =?VE''$
5. $C_0 \dagger \langle 0, 0, 0, 0, ?VE'', 0\rangle =?C'$
6. $?C' = \langle ?T', ?U', ?SE', ?TE', ?VE', ?EE'\rangle$
7. $?VE'(Id) =?\sigma'$
8. $?\sigma' \succ ?\tau$

Using overwrite apply rules, as well as the rule for trivial instances of type schemes, we obtain:

Level 5

$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$

1. $Close(C_0, \{\langle Id, All, 0, FunType, ?\tau', ?\tau'\rangle\}) =?VE''$
2. $C_0 \dagger \langle 0, 0, 0, 0, ?VE'', 0\rangle =?C'$
3. $?C' = \langle ?T', ?U', ?SE', ?TE', ?VE', ?EE'\rangle$
4. $?VE'(Id) =?\sigma'$
5. $?\sigma' \succ ?\tau$

The new first subgoal needs careful handling. When we unravel the various rules for the *Close* operation, we eventually reach the following:

Level 6

$C_0 \vdash$ `let val` $Id =$ `fn` x `=>` x `in` Id `end` $\Rightarrow ?\tau$

1. $tyvars(?\tau') =?A$
2. $tyvars(C_0) =?B$
3. $\langle All, ?B, FunType, ?\tau', ?\tau'\rangle =?\sigma'$

   ... (other subgoals)

n. $?\sigma' \succ ?\tau$

We have now reached a critical point. We can see from subgoals 3 and n that *Id* is going to have a function type of the expected form $?\tau' \rightarrow ?\tau'$. However, to make further progress, the 'unknown' type $?\tau'$ must be instantiated; `elab_tac` does this automatically, replacing it by a type variable such as α! In other words, at this point `elab_tac` infers the polymorphic type for the identity function. This will allow $tyvars(?\tau')$ to be calculated, and the proof can progress.[5] The instantiation gives:

[5] It should be noted that in more complicated cases, where the same polymorphic function is instantiated with several different types within the same expression, the scheme variable may occur elsewhere in a subgoal in such a way that it cannot be instantiated here to a type variable (for example it may end up being a function type). The tactic therefore checks that the scheme variable to be instantiated does not occur as a right hand side of a later subgoal.

Level 7

$C_0 \vdash$ `let val` $Id = $ `fn` $x =>$ x `in` Id `end` $\Rightarrow ?\tau$

1. $tyvars(\alpha) = ?A$
2. $tyvars(C_0) = ?B$
3. $\langle All, ?B, FunType, \alpha, \alpha\rangle = ?\sigma'$

   ....(other subgoals)

n. $?\sigma' \succ ?\tau$

The rest of the proof is straightforward. Using the following facts:

$$tyvars(\alpha) = \{\alpha\}$$
$$tyvars(C_0) = \emptyset$$

along with some trivial set-theory, we eventually have

Level 8

$C_0 \vdash$ `let val` $Id = $ `fn` $x =>$ x `in` Id `end` $\Rightarrow \langle FunType, \alpha, \alpha\rangle$

No subgoals!

This result takes 1.9 seconds to prove in one step, using `elab_tac`.

In Section 10 we discussed the following expression, in which the identity function is applied to itself:

```
let val Id = fn x => x in Id (Id) end
```

. Without working through the automated proof which carries out its type inference, we recall that each occurrence of *Id* in this example is a different instance of the same type scheme. Our proof procedure `elab_tac` correctly deals with this possibility, and automatically gives

Level 1

$C_0 \vdash$ `let val` $Id = $ `fn` $x =>$ x `in` $Id(Id)$ `end` $\Rightarrow \langle FunType, \alpha, \alpha\rangle$

No subgoals!

This proof involves a large number of inferences, taking 2.3 seconds.

12 Conclusions and Suggestions for Future Work

We have described a method for reasoning about operational semantics within the Isabelle theorem prover. The **Elle** system, for reasoning about Standard ML programs, has also been described.

In other work by the authors [30], the role of *denotational* semantics in the verification process was discussed, and a possible mechanisation explored in the HOL system. It is instructive to compare the two approaches. We believe that the present work shows the greater flexibility and usefulness of the *operational* semantics approach as a means of defining the meaning of language constructs.

We believe that the present work will benefit a number of people. It should be of help to beginners trying to understand the Definition of SML, as well as to implementers. It can assist experts who wish to explore possible design changes and extensions to the language, by aiding reasoning about how the various phrases will interact. We also believe that the system will benefit those interested in operational semantics as a basis for program verification, equivalences and transformations.

The **Elle** system is easy to modify, and is quite efficient, because it exploits Isabelle's liking for inference rules, and keeps costly rewriting to a minimum.

The user interface is still under development. Ideally, it should offer the user a range of choices. A minimal interface would be to mimic that of an ML interpreter, presenting only the value and type of the most recently declared variable(s). The maximal interface would be to describe fully the proof of each sequent, showing also the environment (or context) before and after the evaluation (or elaboration). This information can quickly become too much to cope with. Work is progressing on capturing the **Elle** system within the XIsabelle interface mentioned earlier.

Further work needs to be done to extend the system to allow for a larger subset of SML. Exceptions and imperative features will require proper handling of the state and exception conventions [1, 29], while the inclusion of the modules system would allow experiments with subtle aspects of signature matching and elaboration of functors and signature expressions.

Another interesting extension is to general proofs of correctness of algorithms written in SML, for which we need to have an expressive specification language. Since our system is built on Isabelle's version of ZF set theory, we already have the basis of such a language. Such specification constructs are under investigation as part of the Extended ML language of Sannella and Tarlecki [31], and in current work of Gene Rollins, Jeannette Wing, and Amy Moormann Zaremski at CMU on Larch/ML [32].

Reasoning about program equivalences and transformations [33, 34] is also an important line of investigation. The operational semantics rules which define SML have already been expressed in a two-way form which will facilitate such reasoning.

Our methods will apply well to concurrency, where operational semantics is frequently used to give the meaning of language constructs: possibilities include CML [35], CCS [9] and the tasking model of Ada.

Finally, we believe that the approach will be beneficial in reasoning about properties of formal specifications in languages such as Z, whose semantics can be described by a proof theory such as $\mathcal{W}$ [11]. Preliminary investigations have been carried out in this area; however, the task of capturing the semantics of the Z language in a proof assistant remains a formidable task. It is not clear, for example, how to provide a consistent yet simple enough model for Z schemas and schema operations. Schema quantification is especially tricky.

13 Acknowledgment

The authors wish to thank Malcolm Newey (Australian National University), Peter Lindsay and Ed Kazmierczak (Software Verification Research Centre, University of Queensland) for discussions. We also wish to thank the referees, Larry Paulson (University of Cambridge) and Jim Grundy (DSTO), for their careful reading of the paper, and for many useful comments and suggestions.

Bibliography

[1] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[2] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, University of Aarhus, Denmark.

[3] R. Milner. Language Semantics. Notes for Computer Science 3 Course, University of Edinburgh, 1986.

[4] R. Milner. *Calculus of Communicating Systems*. Lecture Notes in Computer Science, No 92. Springer-Verlag, 1980.

[5] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[6] L. C. Paulson. *Introduction to Isabelle, The Isabelle Reference Manual and Isabelle's Object Logics*. Computer Laboratory, University of Cambridge, June 1993.

[7] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science, No 78. Springer-Verlag, 1979.

[8] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[9] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall.

[11] J. C. P. Woodcock and S. M. Brien. W: A Logic for Z. In *Sixth Annual Z User Workshop, York*, 1991.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] C. B. Jones, K.D. Jones, P. A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer-Verlag, 1991.

[14] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[15] D. Craigen et al. EVES: An Overview. Conference Paper CP-91–5402–43, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario, March 1991.

[16] R. Sethi. *Programming Languages, Concepts and Constructs*. Addison-Wesley, 1989.

[17] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[18] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. *Logic and Computer Science (P Odifreddi, ed)*, pages 361–385, 1990.

[19] L. C. Paulson. Set Theory for Verification: I. From Foundations to Functions. II. Induction and Recursion. Report, University of Cambridge, 1993.

[20] D Clement. GIPE: Generation of Interactive Programming Environments. *Technique et Science Informatiques*, 9:157–165, 1990.

[21] L. Théry, Y. Bertot, and G. Kahn. Real Theorem Provers Deserve Real User-Interfaces. In *Proceedings of Fifth Symposium on Software Development Environments*. ACM, 1992.

[22] S. Finn and M. Crawley. *Using Poly/Ml 2.05M.* Abstract Hardware Ltd, Building 2, The Science Park, Brunel University,Uxbridge, Middlesex.UB8 3PQ, U.K., October 1993.

[23] M. A. Ozols. Interface Requirements for Proof Assistants. Divisional Paper ITD-94–04, Information Technology Division, DSTO, 1994.

[24] G. Huet. A Unification Algorithm for Typed $\lambda-$calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[25] J. M. Spivey. *The Z Notation: A reference manual.* Prentice Hall International Series in Compuer Science, 1989.

[26] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, 1986.

[27] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[28] A. Wikström. *Functional Programming Using Standard ML.* Prentice-Hall International Series in Computer Science, 1987.

[29] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[30] A. Cant and M. A. Ozols. The Role of Denotational Semantics in Program Verification. Research Report (to be submitted), Electronics and Surveillance Research Laboratory, DSTO, 1994.

[31] D. T. Sannella and A. Tarlecki. Towards Formal Development of ML Programs: Foundations and Methodology. Report EFS-LFCS-89–71, University of Edinburgh, 1989.

[32] J. Wing, E. Rollins, and A. Moorman Zaremski. Thoughts on a Larch/ML and a New Application for LP. Report CMU-CS-92–135, Carnegie-Mellon University, 1992.

[33] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing Systems*, 20:799, 1990.

[34] R. Roxas and M. C. Newey. Proof of Program Transformations. HOL '91 User Meeting, Aarhus, Denmark, Australian National University, 1991.

[35] J. H. Reppy. *Concurrent Programming with Events: The Concurrent ML Manual.* Dept of Computer Science, Cornell University, Ithaca NY, October 1991.

APPENDIX A
The Theory Tags

The theory Tags is an extension of ZF, designed to meet the need for built-in distinct variables. It defines two new constants, black and white, of meta-type $\mathrm{i} \Rightarrow \mathrm{i}$ (i.e. which take sets to sets) as follows:

$$\mathrm{black(k)} \equiv \langle 0, \mathrm{k} \rangle$$

$$\mathrm{white(k)} \equiv \langle \mathrm{succ}(0), \mathrm{k} \rangle$$

The following rules can easily be derived:

$$\frac{\mathrm{black(m) = black(n)}}{\mathrm{m = n}} \quad (\mathrm{black\_inject})$$

$$\frac{\mathrm{white(m) = white(n)}}{\mathrm{m = n}} \quad (\mathrm{white\_inject})$$

$$\frac{\mathrm{white(m) = black(n)}}{\mathrm{P}} \quad (\mathrm{white\_neq\_black})$$

$$\frac{\mathrm{white(m)} = 0}{\mathrm{P}} \quad (\mathrm{white\_neq\_0})$$

$$\frac{\mathrm{black(m)} = 0}{\mathrm{P}} \quad (\mathrm{black\_neq\_0})$$

For each member id of the set I of identifier classes, we declare id as a set, and posit an injection in_id : $\mathrm{i} \Rightarrow \mathrm{i}$ which satisfies the rules[6]

$$\frac{\mathrm{in\_id(m) = in\_id(n)}}{\mathrm{m = n}} \quad (\mathrm{in\_id\_inject})$$

$$\frac{}{\mathrm{in\_id(m) \in id}} \quad (\mathrm{in\_id\_type})$$

Then, for each proposed variable name of class id, we declare it as a constant, and construct a parse translation which gives, as the internal representation of this variable, a unique term made up of successive applications of the operators black and white. This is done in such a way as to guarantee the mutual distinctness of *all* such defined variables, by means of standard proof techniques using the above rules. Print translations are also needed, to ensure that these terms correctly print back to the expected form. We shall not go into the details of these parse and print translations here.

[6] Note that the rule `in_id_type` should restrict the m to come from some explicitly constructed set, otherwise `id` itself will be "too big" to be constructed in ZF. This would be done most suitably using the theory of Inductive Definitions in Isabelle ZF. However, we have not done this, because the whole Tags theory is quite peripheral to the main **Elle** develoment, and there may be other, simpler, ways to ensure that the desired identifiers are distinct.

THIS IS A BLANK PAGE

DISTRIBUTION

		No. of Copies
Defence Science and Technology Organisation		
Chief Defence Scientist	)	
Central Office Executive	)	1 shared copy
Counsellor, Defence Science, London		Cont Sht
Counsellor, Defence Science, Washington		Cont Sht
Senior Defence Scientific Adviser		1 copy
Scientific Adviser POLCOM		1 copy
Director, Aeronautical & Maritime Research Laboratory		1 copy
Navy Office		
Naval Scientific Adviser		1 copy
Army Office		
Scientific Adviser, Army		1 copy
Airforce Office		
Air Force Scientific Adviser		1 copy
Defence Intelligence Organisation		
Assistant Secretary Scientific Analysis		1 copy
Defence Signals Directorate		
Dr Jeremy Dawson		1 copy
Electronics & Surveillance Research Laboratory		
Director		1 copy
Chief Information Technology Division		1 copy
Chief Electronic Warfare Division		Cont Sht
Chief Guided Weapons Division		Cont Sht
Chief Communications Division		Cont Sht
Chief Land, Space and Optoelectronics Division		Cont Sht
Chief High Frequency Radar Division		Cont Sht
Chief Microwave Radar Division		Cont Sht
Chief Air Operations Division		Cont Sht
Chief Maritime Operations Division		Cont Sht
Research Leader Command & Control and Intelligence Systems		1 copy
Research Leader Military Computing Systems		1 copy
Research Leader Command, Control and Communications		1 copy
Manager Human Computer Interaction Laboratory		Cont Sht
Head, Program and Executive Support		Cont Sht
Head, Command Support Systems Group		Cont Sht
Head, Intelligence Systems Group		Cont Sht
Head, Systems Simulation and Assessment Group		Cont Sht
Head, Exercise Analysis Group		Cont Sht

DISTRIBUTION (CONT)

	No. of Copies
Head, C3I Systems Engineering Group	Cont Sht
Head, Computer Systems Architecture Group	Cont Sht
Head, Information Management Group	Cont Sht
Dr Stephen Crawley, IM Group, Information Technology Division	1 copy
Head, Information Acquisition & Processing Group	Cont Sht
Head, Trusted Computer Systems Group	1 copy
Dr Jim Grundy, TCS Group, Information Technology Division	1 copy
Katherine Eastaughffe, TCS Group, Information Technology Division	1 copy
Dr Tony Cant (Author)	10 copies
Dr Maris Ozols	1 copy
Publications & Publicity Officer ITD	1 copy
Libraries and Information Services	
Australian Government Publishing Service	1 copy
Defence Central Library, Technical Reports Centre	1 copy
Manager, Document Exchange Centre, (for retention)	1 copy
National Technical Information Service, United States	2 copies
Defence Research Information Centre, United Kingdom	2 copies
Director Scientific Information Services, Canada	1 copy
Ministry of Defence, New Zealand	1 copy
National Library of Australia	1 copy
Defence Science and Technology Organisation Salisbury, Research Library	2 copies
Library Defence Signals Directorate Canberra	1 copy
British Library Document Supply Centre	1 copy
Parliamentary Library of South Australia	1 copy
The State Library of South Australia	1 copy
Spares	
Defence Science and Technology Organisation Salisbury, Research Library	35 copies

Department of Defence

DOCUMENT CONTROL DATA SHEET

1. Page Classification
Unclassified

2. Privacy Marking/Caveat
(of document)
N/A

3a. AR Number AR-008-926	3b. Laboratory Number DSTO-RR-0008	3c. Type of Report Research Report	4. Task Number N/A
5. Document Date AUGUST 1994	6. Cost Code N/A	7. Security Classification	8. No. of Pages 56
			9. No. of Refs. 36

7. Security Classification

*U	U	U
Document	Title	Abstract

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L) in document box.

10. Title

AN APPROACH TO AUTOMATED REASONING ABOUT OPERATION SEMANTICS
AL

11. Author(s)

A. Cant and M.A. Ozols

12. Downgrading/Delimiting Instructions

N/A

13a. Corporate Author and Address

Electronics & Surveillance Research Laboratory
PO Box 1500, Salisbury SA 5108

13b. Task Sponsor

N/A

14. Officer/Position responsible for

Security:..........N/A...

Downgrading:......N/A..

Approval forRelease:..CITD..

15. Secondary Release Statement of this Document

APPROVED FOR PUBLIC RELEASE

16a. Deliberate Announcement

No Limitation

16b. Casual Announcement (for citation in other documents)

[✓ No Limitation] ☐ Ref. by Author , Doc No. and date only.

17. DEFTEST Descriptors

Programming languages
Reasoning
Computer program verification
Operational semantics

18. DISCAT Subject Codes

09

19. Abstract

The assurance of the safety or security of critical software rests on a clear understanding of the formal semantics of the programming language used. Operational semantics is the most widely used means of formally defining a language. The need for high levels of assurance, along with the complexity of these definitions for real programming languages, means that tool support is essential for carrying out reasoning about code with respect to the language definition.

In this paper, we describe a generic approach to automated reasoning about the operational semantics of programming languages. As an application of this approach, we describe the construction of an environment for reasoning about programs written in a functional subset of ML. The system we describe (called **Elle**) captures the formal operational semantics definition of a large subset of Standard ML within the theorem prover Isabelle, and provides some support for the verification of ML programs.